Compositional Design of Cyber-Physical Systems Using Contracts

by

Pierluigi Nuzzo

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
Associate Professor Sanjit A. Seshia
Professor James W. Pitman

Summer 2015

Compositional Design of Cyber-Physical Systems Using Contracts

Abstract

Compositional Design of Cyber-Physical Systems Using Contracts

by

Pierluigi Nuzzo

Doctor of Philosophy in Engineering – Electrical Engineering and Computer Sciences

University of California, Berkeley

Professor Alberto L. Sangiovanni-Vincentelli, Chair

The realization of large and complex cyber-physical systems (such as "smart" transportation, energy, security, and health-care systems) is creating design and verification challenges which will soon become insurmountable with the current engineering practices. These highly heterogeneous systems, tightly combining physical processes with computation, communication, and control elements, would substantially benefit from hierarchical and compositional methodologies to make their design possible let alone optimal. Several languages and tools have been proposed over the years to enable model-based development of complex systems. However, an all-encompassing design framework that helps interconnect different tools, possibly operating on different system representations, is still missing.

In this dissertation, we introduce a design methodology that addresses the complexity and heterogeneity of cyber-physical systems by using assume-guarantee contracts to formalize the design process and enable the realization of system architectures and control algorithms in a hierarchical and compositional way. In our methodology, components are specified by contracts, and systems by compositions of contracts. Contracts explicitly define the assumptions of a component on its environment and the guarantees of the component under these assumptions. Contract operations and relations, such as composition, conjunction and refinement allow proving that: (i) an aggregation of components are compatible, i.e. there exists a legal environment in which they can operate; (ii) a set of specifications are consistent, i.e. there exists an implementation satisfying all of them; (iii) an aggregation of components refines a specification, i.e. it implements the specification contract and is able to operate in any environment admitted by it. While horizontal contracts are used to specify components and aggregations of components at the same level of abstraction, we introduce the notion of vertical contracts to reason about richer refinement relations and mappings between different abstraction levels, possibly described by heterogeneous architectures and behavior formalisms. Moreover, we further investigate the problem of compatibility for systems with uncontrolled inputs and controlled outputs, by establishing a link between the theory of contracts and the one of interfaces, which rely on different mathematical formalisms, while sharing the same objectives. From this link, we derive a new projection

operator on contracts that enables the preservation of the semantics of interface composition and compatibility.

Resting on the above contract framework, the design is carried out as a sequence of refinement steps from a high-level specification to an implementation built out of a library of components at the lower level. To allow for requirement analysis and early detection of inconsistencies, top-level system requirements are captured as contracts, by leveraging a front-end pattern-based specification language and a set of back-end formal languages, including mixed integer-linear constraints and temporal logic. Top-level contracts are then refined to achieve independent development of system architectures and control algorithms, by combining synthesis from requirements and optimization methods.

To enable efficient architecture selection under safety and reliability constraints, we explore two optimization-based methods that use an approximate reliability analysis technique to overcome the exponential complexity of exact computations. The Integer-Linear Programming with Approximate Reliability (ILP-AR) method generates larger, monolithic optimization problems using approximate but efficient reliability computations with an explicit theoretical bound on the error. Conversely, the Integer-Linear Programming Modulo Reliability (ILP-MR) method breaks the complex architecture selection task into a sequence of smaller optimization tasks without reliability constraints, interleaved with exact reliability checks. By relying on efficient mechanisms to prune out candidate architectures that are inconsistent with the reliability constraints, ILP-MR can run faster than ILP-AR on large problem instances.

We further explore two methods to systematically design control strategies for a given architecture. The reactive synthesis-based optimal control mapping (RS-OCM) method generates controllers by combining reactive synthesis from linear temporal logic contracts with optimization techniques based on simulation and monitoring of signal temporal logic contracts. Different design concerns are then addressed by leveraging the most appropriate abstraction levels, using contracts from the pre-characterized library to accelerate verification tasks. The programming-based optimal control mapping (P-OCM) method uses, instead, a discrete-time representation of the system and a formalization of the design requirements in terms of arithmetic constraints over real numbers to cast the control problem as an optimization problem over a finite time horizon. The optimization problem is then solved with a receding horizon approach and scales better than monolithic reactive synthesis from linear temporal logic.

We demonstrate, for the first time, the effectiveness of a contract-based design flow on real-life examples of industrial relevance, namely, the design of aircraft electric power distribution and environment control systems. In our framework, optimal selection of large, industrial-scale power system architectures can be performed in a few minutes. Design validation of power system controllers based on linear temporal logic contracts shows up to two orders of magnitude improvement in terms of execution time with respect to conventional techniques. Finally, our optimization-based load management scheme allows better resource utilization than a conventional one.

To my family: Antonietta, Corrado, and Alessandro
To my aunt, Mimina

"General Systems theory should be an important means of instigating the transfer of principles from one field to another (so that it would) no longer be necessary to duplicate the discovery of the same principles in different fields."
Ludwig von Bertalanffy

"Perfection (in design) is achieved, not when there is nothing more to add, but when there is nothing left to take away."
Antoine de Saint-Exupéry

"Everything must be made as simple as possible. But not simpler."
Albert Einstein

Contents

List of Figures

List of Tables

Acknowledgments

This thesis marks the conclusion of an intense academic and research experience, as well as the end of an exciting, unique "adventure" as a student at Berkeley. I try my best to recall the persons I met along my path, and to whom I feel greatly indebted. Certainly, they far outnumber the ones I can mention here.

I wish to first thank my adviser, Alberto Sangiovanni-Vincentelli, for his outstanding, holistic vision, and his invaluable support from both a technical and a human viewpoint. Alberto has been a strong inspiration for my research and its interdisciplinary vocation; he was always available to give me his advice every time I needed it, to carefully review my results, and boost my enthusiasm with his energy.

I thank Prof. Sanjit Seshia and Prof. Jim Pitman, for serving as members of my dissertation committee. I owe to Sanjit my first exposure to formal verification methods and the beauty of the mathematics behind them. Sanjit has been a brilliant guide and collaborator for several key projects during my Ph.D. period, always paying great attention to the progress of my work. I owe to the enthusiasm and guidance of Prof. Jim Pitman and Prof. David Aldous one of the most rewarding intellectual experiences at Berkeley, my introduction to probability theory, a fascinating discipline that has deeply shaped my zest for elegant and rigorous foundations for design methods.

I am deeply grateful to Prof. Kurt Keutzer, for serving as a member of my qualifying exam committee. Kurt was the first professor I met in a class at Berkeley; his welcoming attitude, passion, and enthusiasm will always stay impressed in my mind. As Kurt introduced us to the main algorithms used in electronic design automation, we also started appreciating the durable impact of elegant solutions to concrete design problems.

I have also been fortunate to pursue part of my research within the industrial Cyber-Physical (iCyPhy) systems and the TerraSwarm research centers, which allowed me to interact and collaborate with a unique mix of researchers from both academia and industry. Among the principal investigators of these centers, I would like to thank Prof. Edward Lee and Prof. Richard Murray, for their appreciation of my work and the many interesting and insightful discussions we had. They have also largely inspired my research and provided feedback on some of the results of this thesis. Among the industrial collaborators of the centers, I would like to express my gratitude to Clas Jacobson, Cong Liu, Richard Poisson, Eelco Scholte, Jeff Ernst, Earl Lavallee, Claudio Pinello (United Technology Corporation), Henry Broodney, Yishai Feldman, Amit Fisher, and Michael Masin (IBM), for providing constructive discussions throughout the development of this research. Working with such a team was an enriching and enjoyable experience, and I definitely learned a lot in the different areas of cyber-physical system design, thus developing and refining my skills as a designer and system engineer.

While at Berkeley, I had the opportunity to interact with several other faculty members. Some of them were co-authors of some of my publications; others will have a special place in my mind for having been excellent and passionate instructors. My deepest gratitude goes to Prof. Elad Alon, Prof. Jaijeet Roychowdhury, Prof. Kris Pister, Prof. Martin Wainwright,

Prof. Murat Arcak, Prof. Michael Jordan, Prof. Venkat Anantharam, Prof. Bernhard Boser, Prof. Claire Tomlin, Prof. Stavros Tripakis, Prof. Robert Brayton, Prof. Jan Rabaey, for everything I could learn from them, which contributed to enlarge my cultural and professional horizons. A special thanks to Elad, for the several interactions during my first years at Berkeley, and his invaluable guidance and support as a mentor during my teaching experience. I also thank Stavros for the several insightful and enjoyable discussions we had; his introduction to the theory of relational interfaces was instrumental to some of the theoretical results in this thesis.

I also had the fortune to collaborate with several faculty members in other universities. In particular, I am grateful to Prof. Luca Carloni for always paying a special attention to my work, and for his support and advice along my Ph.D. path and beyond; to Prof. Roberto Passerone, for the insightful and pleasant discussions we had during conferences; to Prof. Tiziano Villa, for being an unconditional source of support and encouragement since I started my Ph.D. at Berkeley. Tiziano has been among the first ones to believe in my academic potential. Moreover, since he visited Berkeley every summer, he was present each time I was about to reach a key Ph.D. milestone, thus sharing with me all the ups and downs along the road.

Berkeley also offered me the opportunity to work at the side of some of the most talented and creative students and colleagues I have ever had. And I am glad that many of them became my friends. A grateful appreciation goes to present and past members of Alberto's group: Mehdi Maasoumy, Liangpeng Guo, Xuening Sun, Nikunj Bajaj, Chung-Wei Lin, John Finn, Baihong Jin, Shromona Ghosh, Antonio Iannopollo, Yang Yang, Qi Zhu, Alessandro Pinto. With all of them I shared some of the most exciting moments and satisfactions of my research. Special thanks also to all the visiting students and scholars I had the pleasure to work with over these years: Michele Lora, Marco Marazza, Lv Chen, Elisabetta Alfonsetti, Safa Messaoud, Piergiuseppe di Marco, Mohammad Mozumdar. Thank you all for the enjoyable moments (and sometimes weekends) spent together in the office, at lunchtime, or at the coffee corner in the lounge.

I have also worked or interacted with several students and visiting scholars in other research teams at Berkeley. I want to extend my gratitude to Baruch Sterin, Yen-Sheng Ho, Yu-Yun Dai, Fabien Chraim, Sayak Ray, Forrest Iandola, Daniel Holcomb, Chenjie Gu, Aadithya Karthik, Prateek Bhanshali, Luigi Di Guglielmo, Indranil Saha, Rohit Sinha, Dorsa Sadigh, Jonathan Kotker, Wei Yang Tan, Ankush Desai, Wenchao Li, Vasu Raman, Tobias Welp, David Burnett, Ben Zhang, Ilge Akkaya, Fabio Cremona, Chris Shaver, Marten Lohstroh, Henrik Ohlsson, Ehsan Elhamifar, Wen Li, John Crossley, Yue Lu, Yida Duan, Chintan Thakkar, Kwangmo Jung, Simone Gambini, Rikky Muller, Armin Wasicek, Patricia Derler, and all the DOP and BWRC students. I have learnt a lot from everyone, and many of them became my friends. Thanks to all the students of the classes I taught; ultimately it was their passion and work that made a successful class. Many thanks also to the exceptional staff at the DOP center, the CHESS center and the EECS department, in particular to Christopher Brooks, Barb Hoversten, Jessica Gamble, Mary Stuart and Shirley Salanio.

Some of the results of my research were strongly influenced by the interaction with Mumu

Xu and Necmiye Ozay at Caltech and Alexandre Donzé at Berkeley. I am very grateful to them for the enriching and extremely productive collaboration. Special thanks also go to Yasser Shoukry, who has more recently contributed to revitalize my interest in decision procedures for convex and nonlinear arithmetic constraints over the reals: our interaction has already spurred several technical contributions.

My experience at Berkeley would have not been the same without the support of all my friends. My first thought goes to those who have lived, together with me, in what we call the "purple house": Marina Romani, Antonio Rosato, and Margherita Ghetti. In them I found loyal and sincere friends, whose aid and encouragement was always welcome. I believe it is because of our long and intense conversations about music, art, history, economics, and life, that I became a better person, rather than just a better scientist. The years I spent at the International House will also have a special place in my mind: an incredible number of friendships started there and are still lasting today. Among others, my sincere gratitude goes to Maria Rosaria Marsico, Andrea Costa, Caterina Migani, Carlo Visconti, and Catia Cialani.

I am very fortunate that many other friends I met the first time in Pisa, at Sant'Anna School, and at IMEC, in Leuven, have still managed during these years to keep in touch with me, even remotely, and to get informed about my trials and tribulations. To name but a few, Vito Giannini, Vincenzo Chironi, Angelo Cerulo, Antonio Molfese, Gabriella Semeraro, Sara Campinoti, Francesco Cusumano, Edoardo Valori, Andrea Barison, Alessio Mazzocco, Giovanni Coluccia, Carlo Michele Petracca, Nicola Nizza, Massimiliano Solazzi, Giacomo Oddo, Luca Martone, Alessandro Natalini, Claudio Nani, Francesca Giraudo, Costantino Armiento. I am deeply grateful to those who visited me: I will never forget the nice time spent together in our trips around California. I am also thankful to those who took their time to organize gatherings when I happened to visit Italy. Moreover, I feel indebted to my friends and former teammates in IMEC for always taking the chance of contacting me and catching up with me every time a conference would bring them to California.

My grateful thought goes finally to my roots and to my friends at home, who never forgot about me. To Sonia Santoro and Alessadro Toma, among them, for their constant and lively interest in my vicissitudes. To my grandparents, my aunts and uncles who have always paid a special attention to me and my personal growth. To aunt Mimina, for her unconditional trust and support for whatever choice I have made in my life, no matter how "risky" or "inconsiderate" it might sound. To my family, for the unquestionable pillar of support they continuously offer. If I have ever accomplished something in my life, it is because I know I can always count on their love. To them I finally dedicate this work, since without them I would not be where I am today.

Chapter 1

Introduction

This chapter provides the motivation for the work in this dissertation, and a preview of the main results. We first discuss some of the challenges for the realization of cyber-physical systems, and introduce an aircraft electric power system as a running example to illustrate them. We then offer an overview of the strategies adopted to address these challenges, and highlight the key technical contributions. Finally, we outline the covered topics and their organization.

1.1 Cyber-Physical System Design Challenges

A large number of new applications are emerging, which go beyond the traditional boundaries between computation, communication and control. The majority of these applications, such as "smart" buildings, "smart" traffic, "smart" grids, "smart" cities, cyber security, and health-care wearables (Figure 1.1), build on *distributed, networked, sense-and-control platforms*, characterized by the tight integration of "cyber" aspects (computing and networking) with "physical" ones (e.g., mechanical, electrical, and chemical processes). In these *cyber-physical systems* (CPS) embedded computers and networks monitor and control the physical processes, usually with feedback loops where physics affects computation and *vice versa* [192, 120, 156, 124].

Intelligent systems that gather, process and apply information are changing the way entire industries operate, and have the potential to radically influence how we deal with a broad range of crucial societal problems. Moreover, as embedded digital electronics becomes pervasive and cost-effective, *co-design* of both the cyber and the physical portions of these systems shows promise of making the holistic system more capable and efficient. Indeed, the availability and cooperation of all the elements of a CPS to fulfill common goals can outperform a system in which such elements are kept separated. However, CPS *complexity* and *heterogeneity*, originating from combining what in the past have been separate worlds, tend to substantially increase the design and verification challenges.

A serious obstacle to the efficient realization of CPS is the inability to rigorously model

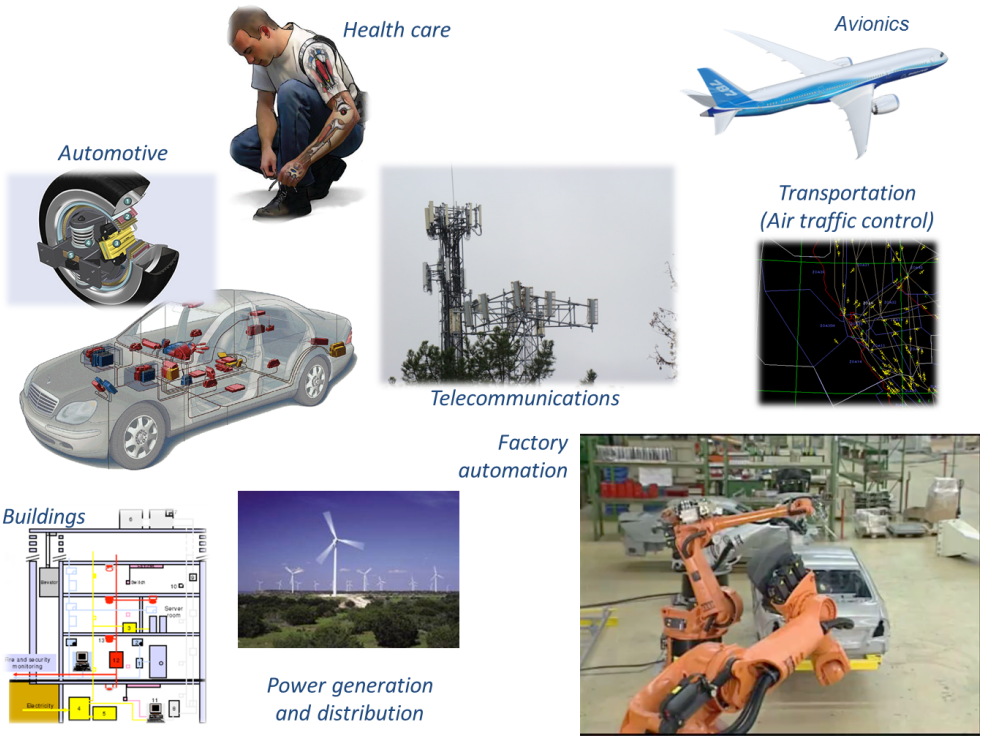


Figure 1.1: Examples of applications of cyber-physical systems. According to the US magazine EE-Times (`http://www.eetimes.com/`), approximately 98% of the world's processors today are not in a PC but "embedded" in a physical system. For instance, a premium car contains around 80 "computers" (electronic control units), 100 million lines of code, and 2 km of wiring (controller area network bus and other networks).

the interactions among heterogeneous components and between the physical and the cyber sides. While in traditional embedded system design the physical system is regarded as a given, the emphasis of CPS design is instead on managing dynamics, time, and concurrency by *orchestrating* networked, distributed computational resources *together* with the physical systems. Functionality in CPS is provided by an ensemble of sensing, actuation, connectivity, computation, storage and energy. Therefore, CPS design entails the convergence of several sub-disciplines, and tends to stress all existing modeling languages and frameworks, which are hardly interoperable today. In computer science, *logic* is emphasized rather than dynamics, and processes follow a *sequential* semantics; computer scientists mostly deal with computational aspects and carefully abstract the physical world. Conversely, *physical processes* are generally represented using continuous-time *dynamical* models, often expressed as differential equations, which are acausal, *concurrent* models; control, electrical and mechanical engineering have to directly deal with the physical quantities involved in the design process. It is, therefore, difficult to accurately capture the interactions between these two

worlds.

Moreover, a severe limitation in common design practice is the lack of *formal specifications*. Requirements are written in languages that are not suitable for mathematical analysis and verification. Assessing system correctness is then left for simulation and, later in the design process, prototyping. Thus, the traditional heuristic design process based on informal requirement capture and designers' experience can lead to implementations that are inefficient and sometimes do not even satisfy the requirements, yielding long re-design cycles, cost overruns and unacceptable delays. The cost of being late to market or of product malfunctioning is staggering as witnessed by the recent recalls and delivery delays that system industries had to bear. Toyota's infamous recall of approximately 9 million vehicles due to the sticky accelerator problem[1], Boeing's 787 delay bringing an approximate toll of $3.3 billion[2] are examples of devastating effects that design problems may cause.

In the remainder of this section, we further detail the difficulties highlighted above, and motivate the research developed in this dissertation. While we build on the seminal elaborations by Derler et al. [72] and Sangiovanni-Vincentelli et al. [177], we offer a new vista of the main CPS design challenges from the perspective of the *system engineers* who are in charge of realizing them. We classify the main issues into three categories: modeling, specification, and integration.

1.1.1 Modeling Challenges

Model-based design (MBD) [193, 179] is today generally accepted as a key enabler for the design and integration of complex systems. However, as mentioned above, because CPS tend to stress all existing modeling languages and frameworks, a set of modeling challenges stem by the difficulty in accurately capturing the interactions between them. We categorize these challenges in terms of: (1) modeling timing and concurrency and (2) modeling the interactions between functionality and implementation.

1.1.1.1 Challenge 1 – Modeling Timing and Concurrency

A first set of technical challenges in the analysis and design of the real-time embedded software in CPS stems from the need to bridge its inherently sequential semantics with the intrinsically concurrent physical world. All the general-purpose computation and networking abstractions are built on the premise that execution time is just an issue of performance, not correctness. Therefore, *timing* of programs is not repeatable, except at very coarse granularity, and programmers have hard time to specify timing behaviors within the current programming abstractions. Moreover, *concurrency*, is often poorly modelled. Concurrent software is today dominated by threads, performing sequential computations with shared memory. Incomprehensible interactions between threads can be the sources of many problems, ranging from deadlock and scheduling anomalies, to timing variability, nondeterminism,

[1] see, e.g., `http://www.autorecalls.us`

[2] see, e.g., `http://en.wikipedia.org/wiki/Boeing_787`

buffer overruns, and system crashes [122]. Finally, modeling *distributed systems* adds to the complexity of CPS modeling by introducing issues such as disparities in measurements of time, network delays, imperfect communication, consistency of views of system state, and distributed consensus [72].

1.1.1.2 Challenge 2 – Modeling Interactions between Functionality and Implementation

Functional models are particularly suitable for prototyping control and data processing algorithms, since they are able to abstract unnecessary implementation details, and can be evaluated more efficiently. However, computation and communication do take time. Therefore, to correctly evaluate a CPS model, it is necessary to also model the dynamics of software and networks. While implementation is largely orthogonal to functionality and should, therefore, not be an integral part of a model of functionality, pure functional models tend to be inaccurate, in that they implicitly assume that data are computed and transmitted in zero time, so that the dynamics of the software and networks have no effect on the system behaviors.

It is then essential to provide mechanisms to capture the interactions of functionality and implementation, while still preserving their separation. Specifically, it should be possible to conjoin two distinct representation of design with each other, namely a functional model and an *implementation model*. The latter allows for design space exploration, while the former supports the design of control strategies. The conjoined models enable evaluation of interactions across these domains.

1.1.2 Specification Challenges

Depending on application domains, up to 50% of all errors result from imprecise, incomplete, or inconsistent and thus unfeasible requirements. The overall system product specification is somewhat of an art today, since to verify its completeness and its correctness there is little that it can be used to compare with. We categorize the specification challenges in terms of: (3) capturing system requirements and (4) managing them.

1.1.2.1 Challenge 3 – Capturing System Requirements

Among the many approaches taken in industry for getting requirements right, some of them are meant for initial systems requirements, mostly relying on ISO 26262[3] compliant approaches. To cope with the inherently unstructured problem of completeness of requirements, industry has set up domain- and application-class specific methodologies, including *learning* processes, such as the one employed by Airbus to incorporate the knowledge base of external hazards from flight incidents. *Use-case analysis* methods as advocated for UML[4]-based

[3] http://www.iso.org/iso/catalogue_detail.htm?csnumber=43464

[4] http://www.omg.org/spec/UML/

development processes [45] follow the same objective. A common theme of these approaches is the intent to systematically identify those aspects of the environment of the system under development whose observability is necessary and sufficient to achieve the system requirements. However, the most efficient way of assessing completeness of a set of requirements is by executing it, which is only possible if *semi-formal* or *formal specification languages* are used, where the particular shape of such formalizations is domain dependent.

1.1.2.2 Challenge 4 – Managing Requirements

Design specifications tend to move from one company (or one division) to the next in non-executable and often unstable and imprecise forms, thus yielding misinterpretations and consequent design errors. In addition, errors are often caught only at the final integration step as the specifications were incomplete and imprecise; further, nonfunctional specifications (e.g., timing, power consumption, size) are difficult to trace.

It is common practice to structure system level requirements into several "chapters," "aspects," or "viewpoints," quite often developed by different teams using different skills, frameworks, and tools. However, these *viewpoints*, e.g., including function, safety, timing, energy, are not unrelated. Without a clean approach to handle multiple viewpoints, as also discussed in Section 1.1.1, the common practice today is to discard some of the viewpoints in a first stage, e.g., by considering only functions and safety. Designs are then developed based on these only viewpoints. Other viewpoints are subsequently taken into account (e.g., timing, energy), thus resulting in late and costly modifications and re-designs.

Requirement engineering is a discipline that aims at improving the situation described above by paying close attention to the management of the requirement descriptions and their traceability (e.g., using commercial tools such as DOORS[5] in combination with REQTIFY[6]) and by inserting, whenever possible, precise formulation and analysis methods and tools. However, the support of formal approaches for requirement structuring and analysis is still largely missing.

1.1.3 Integration Challenges

CPS integrate diverse subsystems by often composing pieces that have been pre-designed or designed independently by different groups or companies. This is done routinely, for example, in the avionics and automotive sectors, albeit in a heuristic and *ad hoc* way. In fact, integrating component models to develop holistic views of the system becomes very challenging. We summarize below the main integration challenges by categorizing them in terms of: (5) preventing misconnected model components, (6) keeping model components consistent, and (7) improving scalability and accuracy of model analysis.

[5] http://www-03.ibm.com/software/products/en/ratidoorfami

[6] http://www.3ds.com/products-services/catia/capabilities/requirements-engineering/reqtify/

1.1.3.1 Challenge 5 – Preventing Misconnected Model Components

The bigger a model becomes, the harder it is to check for correctness of connections between components. Typically, model components are highly interconnected, and the possibility of errors increases. Errors may be due to different units between a transmitting and a receiving port (unit errors), different interpretation of the exchanged data (semantic errors), or just reversed connections among ports (transposition errors). Since none of these errors would be detected by a type system, specific measures should be enabled to automatically check for them [72].

1.1.3.2 Challenge 6 – Keeping Model Components Consistent

Inconsistency may arise when a simpler (more abstract) model evolves into a more complex (refined) one, where a single component in the simple model becomes multiple components in the complex one. Moreover, non-functional aspects such as performance, timing, power, or safety analysis are typically addressed in dedicated tools using specific models, which are often evolved independently of the functional ones (capturing the component dynamics), thus also increasing the risk of inconsistency.

In a modeling environment, a mechanism for maintaining model consistency is needed to allow components to be copied and reused in various parts of the model while guaranteeing that, if later a change in one instance of the component becomes necessary, the same change is applied to all other instances that were used in the design. Additionally, more sophisticated mechanisms would be needed to maintain consistency between the results of specialized analysis and synthesis tools operating on different representations of the same component.

1.1.3.3 Challenge 7 – Improving Scalability and Accuracy of Model Analysis

As stated above, it is essential that the fundamental steps of system design (functional partitioning, allocation on computational resources, integration, and verification) be supported across the entire design development cycle and across different disciplines. CPS may be modeled as hybrid systems integrating solvers that numerically approximate the solutions to differential equations with discrete models, such as state machines, dataflow models, synchronous-reactive models, or discrete event models [123]. A survey of languages and tools for the specification and analysis of CPS models can be found in Chapter 2. However, a major set of challenges for CPS integration is the inadequacy of traditional analysis techniques and their interoperability. In particular, conventional verification and validation techniques tend not to scale to highly complex or adaptable systems (i.e., those with large or infinite numbers of possible states or configurations). On the other hand, simulation techniques may also be affected by modeling artifacts, such as solver-dependent, nondeterminate, or Zeno behaviors [72].

As a concrete example of industrial CPS which exposes several of, if not all, the challenges discussed above, we introduce an *aircraft electric power system* (EPS) in Section 1.2. This

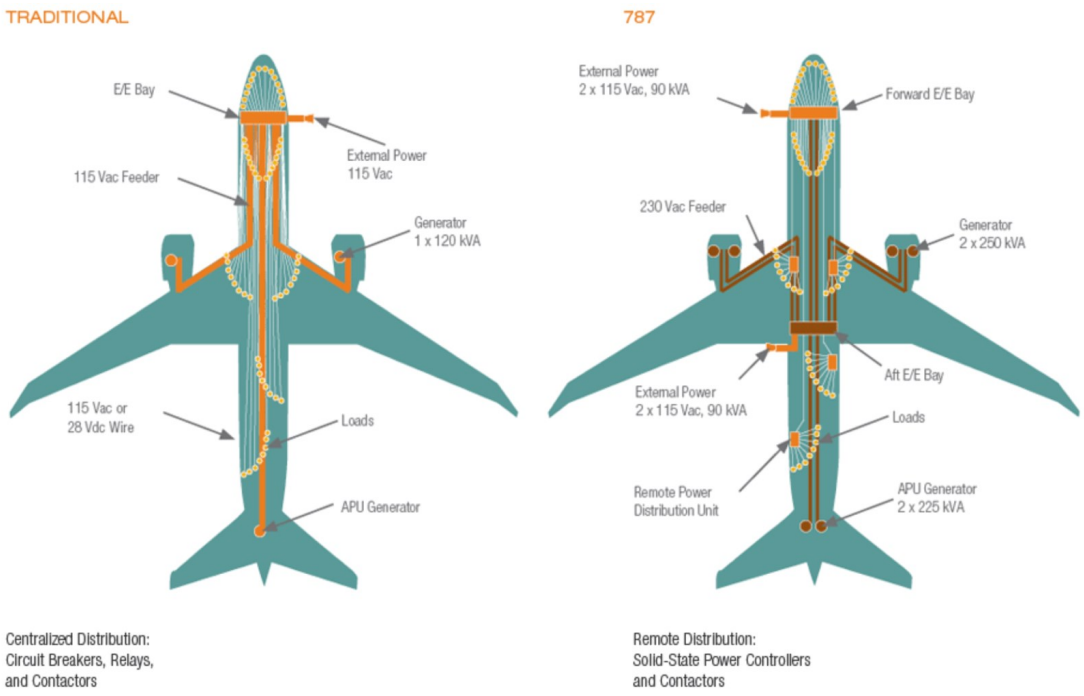


Figure 1.2: Evolution in aircraft electric power system architectures: from traditional architectures (*circa* World War II, left side) to more recent ones (Boeing 787, *circa* 2007, right side). The centralized distribution scheme, relying on mechanical circuit breakers and relays, has been replaced by a remote distribution scheme, extensively relying on solid-state power controllers. Courtesy of United Technology Corporation (UTC), industrial partner of the industrial Cyber-Physical (iCyPhy) system consortium (`http://www.icyphy.org/`).

is also used as a running example to illustrate the methodology and tools developed in this dissertation.

1.2 Running Example: Aircraft Electric Power System Design

The advent of high capability, reliable power electronics together with powerful embedded processors has enabled an increasing amount of "electrification" of vehicles such as cars and

aircraft in recent years [147, 107]. Hydraulic, pneumatic, and mechanical systems are being replaced by cyber-electrical components that decrease weight and increase the overall system efficiency [175]. For example, Figure 1.2 shows the evolution of the architecture of an electric power system in a "more electric" aircraft. Traditional architectures (*circa* World War II), based on centralized distribution and mechanical circuit breakers and relays, have been replaced with more advanced ones (Boeing 787, *circa* 2007), based on remote distribution and solid-state power controllers. As this trend is common to almost all the subsystems of an aircraft, the increased use of electrically-powered elements poses even more challenges to the power system in terms of the *reliability* of electrical power generation and distribution while satisfying *safety* requirements.

In the following, we provide details on the power generation and distribution system in a passenger aircraft, using the sample EPS architecture in Figure 1.3, in the form of a single-line diagram (SLD) [147], a simplified notation for three-phase power systems. We then summarize the system requirements and expose the main design challenges in its realization.

Typically, aircraft electric power systems consist of generation, primary distribution and secondary distribution sub-systems. One or more *supervisory control* units actuate a set of electromechanical switches to dynamically distribute power from generators to loads, while satisfying safety, reliability and real-time performance requirements. In this example, we focus on the primary power distribution system, which includes the majority of the supervisory control logic, and involves the configuration of the contactors that deliver power to high-voltage AC and DC buses and loads.

1.2.1 Components

The main components of an electric power system are generators, contactors, buses, and loads. Primary *generators* are connected to the aircraft engine and can operate at high or low voltages. *Auxiliary generators* are mounted atop an *auxiliary power unit* (APU). The APU is normally used on ground (when no engines are available) to provide hydraulic and electric power, but can also be used in flight when one of the primary generators fails. With a small abuse of notation, we also refer to auxiliary generators themselves as APUs. *Batteries* are primarily used at start-up and in case of emergency. AC and DC *buses* (both high and low-voltage) deliver power to a number of loads. Buses can be essential or non-essential. Essential buses supply loads that should always be powered, while non-essential ones supply loads that may be shed in the case of a fault or limited power capacity.

Contactors are electromechanical switches that connect components, and therefore determine the power flow from sources to loads. They are configured to be open or closed by one or multiple controllers (not shown in Figure 1.3), denoted as *Bus Power Control Units* (BPCU).

Loads include subsystems such as lighting, heating, avionics and navigation. Bus loads also include power conversion devices: *Rectifier Units* (RUs) convert AC power to DC power, AC *transformers* (ACTs) step down a high-voltage to a lower one, *Transformer Rectifier Units* (TRUs) both decrease the voltage level and convert it from AC to DC.

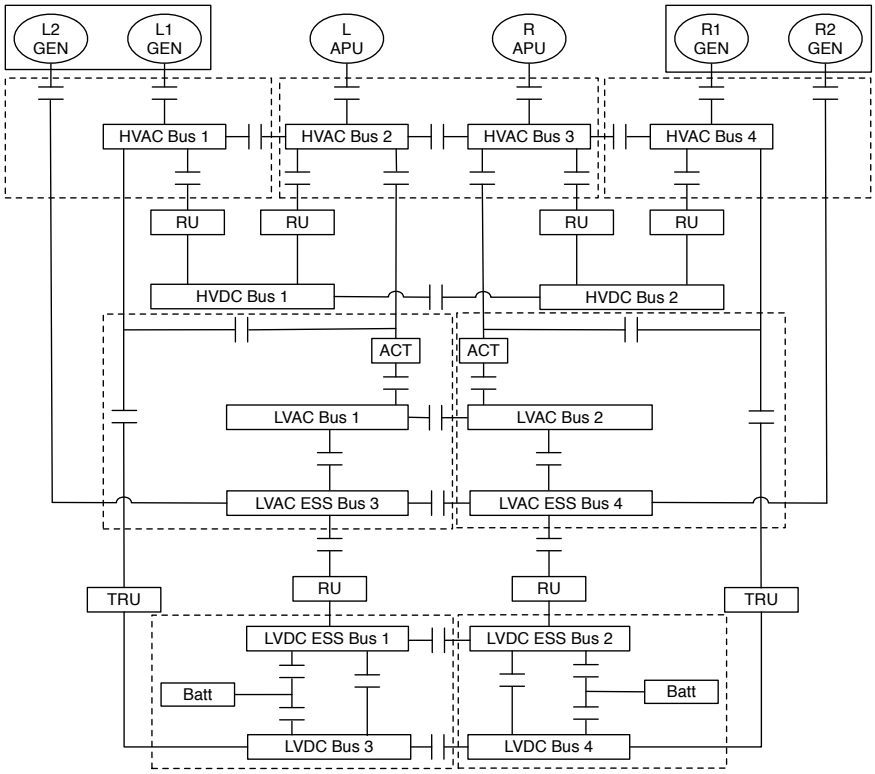


Figure 1.3: Single-line diagram of an aircraft electric power system adapted from a Honeywell, Inc. patent [144] (figure from [154]).

1.2.2 System Description

The main AC power sources at the top of Figure 1.3 include two low-voltage generators, two high-voltage generators, and two APU-mounted auxiliary generators. Each engine connects to a high-voltage AC (HVAC) generator (L1 and R1) and a low-voltage AC (LVAC) generator (L2 and R2). Panels, denoted as dashed square boxes, represent groups of components that are physically separated on the aircraft. The three panels below the generators include the HVAC buses, which can be selectively connected to the HVAC generators, to the auxiliary generators, and to each other via contactors, denoted by double bars.

Four rectifier units are selectively connected to buses as HVAC loads. The two panels below the high-voltage DC (HVDC) buses include the LVAC subsystem. A set of AC transformers (ACTs) convert HVAC power to LVAC power and are connected to four LVAC buses. LVAC ESS Bus 3 and LVAC ESS Bus 4 are essential and are selectively connected to the two low-voltage generators. The LVAC essential buses are also connected to rectifier units, and thus to low-voltage DC (LVDC) power. The LVDC subsystem also contains two batteries. Power can be selectively routed directly from the HVAC bus to the LVDC buses 3 and 4 using TRUs.

One or more bus power control units use sensors (which are not depicted in Figure 1.3) to measure physical quantities, such as voltages and currents, and control the state (open or closed) of the contactors, to dynamically reconfigure the system based on the status and availability of the power sources. For the rest of the thesis, we also denote this centralized or distributed supervisory control unit as BPCU.

Each generator is also controlled by a Generator Control Unit (GCU), which is considered an internal component, and is not explicitly represented in Figure 1.3. The GCU regulates the output voltage level delivered by each generator to be within a specified range. Therefore, fluctuations in the power required by the loads can be directly handled by the GCU within the generator's power rating. On the other hand, whenever the power demand exceeds the generator's capability, the BPCU is responsible for possibly shedding unessential loads or rerouting some of them to another power source.

1.2.3 System Requirements

Given a set of loads, together with their power and reliability requirements, the goal is to determine the system's architecture and control such that the demand of the loads is satisfied for all flight conditions and a set of predetermined faults. To better formalize this design objective, we begin with a qualitative analysis of the main system requirements, by categorizing them in terms of safety, reliability and performance requirements. For each of these categories, we provide a few examples that serve as a reference for the rest of the dissertation.

Safety specifications constrain the way each bus must be powered to avoid loss of essential features, and the maximum time interval allowed for power shortages. For instance, to avoid generator damage, we proscribe AC sources to be paralleled, i.e. no AC bus can be powered by multiple generators at the same time. Moreover, we refine the definition of essential loads and buses (such as flight-critical actuators) provided above by requiring that they be never unpowered for more than a specified time t_{max}.

Reliability specifications describe the bounds on the failure probabilities that can be tolerated for different portions of the system. Based on its failure modes, every component is characterized by a failure rate. A failure rate of λ indicates that a failure occurs, on average, every $1/\lambda$ hours. For a given mission profile, failure rates can be translated into failure probabilities so that system reliability specifications are also expressed in terms of the failure probabilities of the components. Based on the component failure rates, a typical specification would require that the failure probability for an essential load (i.e., the probability of being unpowered for longer than t_{max}) be smaller than 10^{-9} per flight hour. The actual probability value depends on the load criticality [147]. In our example, both the electric power system topology and the controller should be designed to accommodate any possible combination of faults potentially causing the failure of an essential component, and having a joint probability larger than 10^{-9} per flight hour.

Performance requirements specify quality metrics that are desired for the system, in addition to the safety and reliability requirements reviewed above. For instance, to improve

the system performance in response to a failure, each bus may be assigned a priority list determining in which order available generators should be selected to power it. If the first generator in the list is unavailable, then the bus will be powered by the second generator, and so on. A hypothetical prioritization list for the HVAC Bus 1 in Figure 1.3 would require, for instance, that L1 GEN has the priority, if available. Otherwise, Bus 1 should receive power from the R1 GEN, then from the L APU, and finally from the R APU. In a similar way, load management policies are also based on priority tables requiring, for instance, that the available power be first allocated to the non-sheddable loads and then to the sheddable loads, in a prescribed order. In general, bus power priorities can be integrated in the BPCU control logic, while load shedding priorities are handled by a load management controller.

Altogether, an aircraft electric power system, as the one discussed above, offers a concrete example of an industrial-scale, heterogeneous cyber-physical system, suitable enough to illustrate the main concepts of this dissertation, and experiment with the proposed design techniques. Typical sub-systems in aeronautics, such as the EPS, may easily reach a few thousands top-level requirements, often ambiguously expressed using textual languages, and intrinsically heterogeneous, spanning safety, reliability and real-time performance, as highlighted above. As in any CPS design problem, system engineers are to define both the *system architecture*, including the number and type of system components, their dimensions, and their interconnections, and the *control algorithm*, possibly exploring trade-offs across their boundaries. However, as the complexity of this system increases, it is more difficult to perform design space exploration and trade-off analysis at the system level. Designers are expected to solve combinatorial problems over a large, discrete variable space that is coupled to a continuous space, where expensive, high-fidelity simulations must be run to achieve the desired accuracy and provide strong guarantees on the satisfaction of the requirements.

Current design flows for aircraft EPS are, therefore, experiencing all the severe limitations discussed in Section 1.1, because of the lack of formalized specifications, and the inability to rigorously model the interactions among heterogeneous components and between the physical and the cyber sides of the system. To reduce expensive re-design steps, the design is generally addressed by minor incremental changes on top of consolidated solutions. A more systematic approach is hindered by the lack of rigorous design methodologies that allow estimating the impact of earlier design decisions on the final implementation. In the following, we summarize the limitations of current design methodologies in addressing the challenges discussed in Section 1.1, and the strategy proposed in this thesis to overcome them.

1.3 CPS Design Methodology and Tools: The Challenge of Combining Heterogeneous Worlds

Several languages and tools have been proposed over the years to overcome the limitations discussed in Section 1.1 and Section 1.2, provide support for different design tasks, and en-

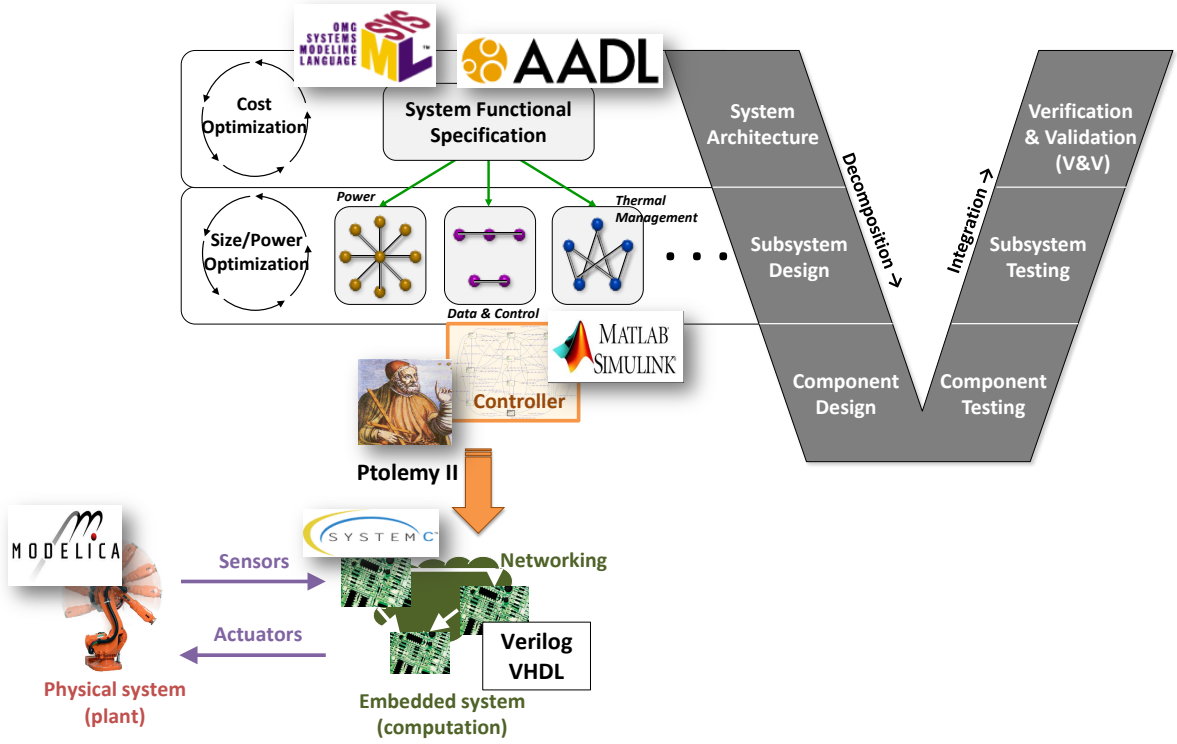


Figure 1.4: Simplified representation of the V-model together with some state-of-the-art languages and tools providing support for different design tasks.

able model-based development of CPS. An overview of some of these languages and tools, which are also iconically represented in Figure 1.4, is provided in Chapter 2. However, the largest benefits in design technologies are deemed to arrive by addressing the entire system design process, rather than just considering point solutions of tools and models that ease only part of the design. Moreover, as we mentioned before, a major bottleneck in the design of cyber-physical systems is the inability to foresee the impact of design decisions made early in the design process, e.g. during the *concept design* phase, on the final implementation. While researchers from both academia and industry have chartered the field of *design methodologies* with increasing clarity, an all-encompassing framework for CPS design that helps interconnect different tools, possibly operating on different system representations, is very difficult to assemble, and most designers still resort to patched flows [156].

Some industrial domains such as automotive and aerospace use the *"V-model"* that was proposed several years ago by the German defense companies[7]. As shown in Figure 1.4, in this methodology, there is a top-down *design process* that ends with system decomposition

[7] `http://v-modell.iabg.de/`

(the left arm of the V) followed by an *integration and verification process* that ends with the verification of the entire system (the right arm of the V). Specifically, as summarized by Benveniste et al. [34], following product level *requirement analysis*, subsequent steps would first evolve a *functional architecture* supporting product level requirements. Sub-functions are then re-grouped taking into account re-use and product line requirements into a *logical architecture*, whose modules can be developed independently, e.g., by different subsystem suppliers. The realization of such modules often involves mechatronic design. The top-level of the technology-oriented architecture would then show the mechatronic architecture of the module, defining interfaces between the different domains of mechanical, hydraulic, electrical, and electronic system design. Subsequent phases would then unfold the detailed design for each of these domains, such as the design of the electronic subsystem involving, among others, the design of electronic control units (ECUs). These design phases are paralleled by integration phases along the right-hand part of the V, such as integrating basic and application software on the ECU hardware to actually construct the electronic control unit, integrating the complete electronic subsystems, integrating the mechatronic subsystem to build the module, and integrating multiple modules to build the complete product. An integral part of V-based development processes are *testing* activities, where at each integration level test-suites developed during the design phases are used to verify compliance of the integrated entity to the specification.

This presentation is overly simplistic, since it does not directly reflect the multi-site, multi-domain, and cross-organizational design teams involved in the design of electronic components in today's complex systems, as well as the parallelization of design activities motivated by the partitioning of the design space into different subsystems and domains. Moreover, re-use strategies lead to separate design activities, which then short-cut or significantly reduce the effort both in design and integration steps. Therefore, in spite of being very popular and widely referenced, the V-model tends to hide the complexity of the actual design processes that system companies develop by themselves. The sequential process that starts with a specification and moves along the arms of the V may often be replaced, in practice, by a number of iterations and "out-of-order" executions of activities. Furthermore, it is often observed that heuristic design processes largely based on the V-model tend to become soon inadequate in many ways:

- This water-fall methodology produced good results when the complexity of the designs was relatively small. When complexity scales up, we cannot simply wait to initiate the verification phase after the design is completed. Conventional *Verification and Validation* (V&V) techniques performed too late in the design flow do not scale to highly complex or adaptable systems. Rather we should favor early verification and continuous monitoring of the design while the refinement steps are taken. In addition, we should favor *"formality"* in all aspects of the design flow to allow analysis and even synthesis with guaranteed properties of the final outcome of the process.

- In traditional flows, *design-space exploration* is rarely performed adequately, yielding suboptimal designs where the architecture selection phase does not consider extensibil-

ity, re-usability, and fault tolerance to the extent that is needed to reduce cost, failure rates, and time-to-market. System-level design exploration is usually the realm of experienced architects, often relying on their accrued knowledge and a set of heuristic evaluations to take risky decisions. Rather, we should favor mechanisms to generate *reliable abstractions* that enable design space exploration across different domains in a scalable way.

- Even if model-based design techniques are largely adopted at the module and component levels, tools are domain-specific and hardly interoperable. We should instead favor *new modeling approaches* that can mix different physical systems, control logic, and implementation architectures. In doing so, existing approaches, models, and tools should be subsumed and not eliminated in order to be smoothly incorporated in current design flows. A design platform should then be developed to host the new techniques and to integrate a set of today's poorly interconnected tools.

In the lack of a comprehensive framework for early requirement validation with tight safety, reliability and performance guarantees, and for scalable, system-level design exploration under a set of heterogeneous constraints, the problem of designing planetary-scale CPS appears insurmountable unless bold steps are taken to bridge the existing gap between *system science* and *system engineering*.

By reflecting on the history of achievements of electronic design automation in taming the design complexity of VLSI systems, we advocate that CPS design automation efforts are doomed to be impractical and poorly scalable, unless they are framed in *structured design methodologies* and in a formalization of the design process in a *hierarchical* and *compositional* way. *Design methodologies* relying on sets of modeling abstractions and related tools to support analysis and synthesis have been instrumental, in the past, to cope with complexity in VLSI design. As major productivity gains are needed today in the design of complex systems, and better verification and validation is a necessity as the safety and reliability requirements become more stringent, design methodology and tools start to be also on the critical path to CPS design [177]. *Hierarchy* has been instrumental to scalable VLSI design, where boosts in productivity have always been associated with a rise in the level of abstraction of design capture, from transistor to register transfer level (RTL), to systems-on-chip. On the other hand, designers typically assemble large and complex systems from smaller and simpler components, such as pre-designed intellectual property (IP) blocks. Therefore, *compositional approaches* offer a "natural" perspective that should inform the whole design process, starting from its earlier stages [151].

Decomposition and *abstraction* are indeed key strategies to manage the design complexity and reduce the number of items to consider, by either breaking the design object into semi-independent parts (divide-and-conquer approach), or by aggregating objects and eliminating unnecessary details with respect to the goal at hand. However, while system engineers routinely make use of decompositions, abstractions and approximations to assemble their designs, system and computer scientists are lagging behind in their quest for the formal

foundations of these concepts and their algorithmic implications in a heterogeneous context. Advancing our understanding of the intricacies of compositional reasoning, and its interplay with abstraction and approximation mechanisms, is therefore at the heart of any effort towards a rigorous design discipline.

1.4 Dissertation Overview

This dissertation is centered on the following thesis:

A contract-based approach provides a formal foundation for a compositional and hierarchical design flow for cyber-physical systems, covering both horizontal and vertical compositions. The theoretical results, methods, and algorithms in this dissertation enable such a flow, by formalizing horizontal and vertical contracts *for different levels of abstraction and viewpoints in the design, and by establishing consistency, compatibility and various refinement relations between these contracts, including synthesis and optimization-based techniques. The overall concepts are demonstrated on the design of aircraft electric power distribution and environment control systems.*

This dissertation presents a path towards an integrated framework for CPS design; the pillars for the framework are a methodology that relies on the *platform-based design* paradigm (PBD) [176] and the algebra of *contracts* to formalize the design process and enable the realization of systems in a hierarchical and compositional manner.

Contracts are mathematical abstractions, explicitly defining the assumptions of each component on its environment and the guarantees of the component under these assumptions. By hiding the internal details of each component while exposing only the relevant information about its interface, contracts often offer a more compact representation of a design. Moreover, by emphasizing the interface between each component and its environment, they provide a disciplined way to perform compositional reasoning, by establishing mechanisms to infer global (system) properties out of local (component) properties. Contracts are then a key *modeling paradigm* to build reliable abstractions of complex systems in a modular and hierarchical way, with the potential to originate verification and synthesis algorithms that are more efficient.

Contract-Based Design (CBD) can indeed be seen as a unifying, formal, compositional paradigm for system design, relying on a rich algebraic framework that can support every step of the design flow. As shown in Figure 1.5, contracts allow structuring and formalizing system requirements, usually as a *conjunction* of different aspects or *viewpoints*, capturing different design concerns (e.g. functional, safety, timing). We can then verify whether a set of specifications are consistent, i.e. there exists an implementation satisfying all of them. Contract *composition* allows reasoning about aggregations of heterogeneous components (e.g. physical, embedded and controller subsystems), and checking whether they are compatible, i.e. there exists a legal environment in which they can operate. Finally, contract

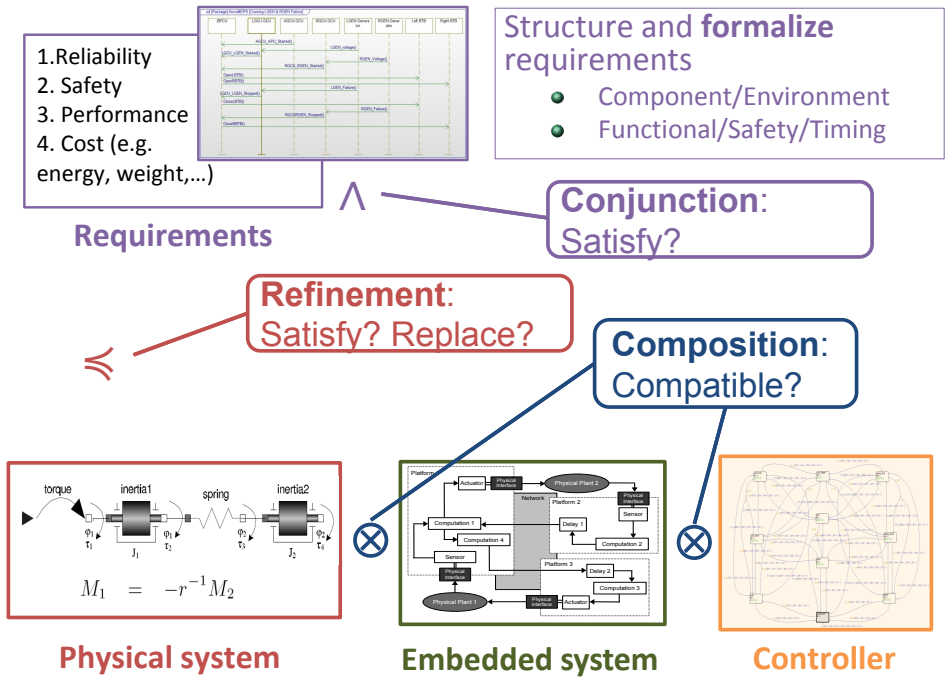


Figure 1.5: Contract-Based Design as a unifying, formal, compositional paradigm for system design.

refinement allows querying whether an aggregation of components refines a specification, i.e. it implements the specification contract and is able to operate in any environment admitted by it. Altogether, we adopt a contract-based framework as a "natural" candidate to lay out the theoretical foundations of our cyber-physical system design tools, with the potential of facilitating a set of design automation efforts, including the creation and exchange of model libraries, and the integration of multiple formalisms, languages and tools.

We then introduce a *design methodology* that uses this assume-guarantee contract framework to address the complexity and heterogeneity of cyber-physical systems, and enable the realization of system architectures and control algorithms in a hierarchical and compositional way. In our methodology, pictorially represented in Figure 1.6, components are specified by contracts, and systems by compositions of contracts. The design is carried out as a sequence of refinement steps from a high-level specification to an implementation built out of a library of components at the lower level. The top-level specification is first formalized in terms of contracts to enable requirement validation and early detection of inconsistencies. Then, at each step, contracts are refined by combining synthesis, optimization and simulation-based design space exploration methods. Contracts are used for the first time in this disserta-

tion to provide a unifying formalization for the *vertical refinement* steps above, and guide their execution using a combination of existing techniques with newly developed ones. The effectiveness of our approach is demonstrated on real-life examples of industrial relevance, namely the design of aircraft electric power distribution and environment control systems. In the following, we detail the main contributions of this thesis, and its organization.

1.5 Main Contributions

This dissertation provides contributions to the areas of contract-based design, requirement engineering, and design space exploration for cyber-physical systems. Based on their nature, we can categorize these contributions as follows.

1.5.1 Theory: Formalisms for Compositional System Design

1.5.1.1 Relation between contract and interface theories

This dissertation pioneers a new research direction, meant to shed light on the effectiveness of different contract, interface and specification frameworks (or their combination) for requirement formalization and validation, early detection of integration errors, and principled use of abstraction-refinement in system design. Rather than relating different theories via a higher-level meta-theory, as proposed by Benveniste et al. [34], our approach aims at finding transformations that can draw direct links between different theories. It is then possible to analyze differences and correspondences between key operators and relations in different theories (i.e. composition, refinement and conjunction) by studying their preservation properties under the proposed transformations. Ultimately, our goal is to assess which formalisms, or combination of formalisms, should be used for different steps in the proposed design flow.

Specifically, in Chapter 3, we apply the approach described above to propose a "natural" transformation between assume-guarantee contracts [33, 34] and synchronous relational interfaces [197]. We then show that refinement is preserved under the transformation, but this is not generally the case for serial composition and conjunction. This investigation finally leads to the definition of a new assumption-projection operator that enriches the contract framework adopted in this thesis, and enables the preservation of the semantics of interface composition and compatibility.

1.5.1.2 Vertical contracts for design refinement

Contracts are used for the first time in this dissertation to provide a unifying formalization for several *vertical refinement* relations that are essential in multi-level and multi-aspect design, encompassing synthesis from requirements, optimization-based refinement, and optimized mapping between abstraction levels. Specifically, in Chapter 3, we introduce the notion of *heterogeneous refinement* to extend the notion of contract refinement between

two abstraction levels specified by different formalisms. Moreover, we generalize the notion of *vertical contracts*, previously introduced in the context of analog and mixed-signal design [152, 155] and embedded software for automotive applications [34], and informally extended to a platform-based design approach in a control setting [177], to the broader context of cyber-physical systems.

Traditionally contracts have been used to specify components, and aggregation of components at the same level of abstraction; for this reason we refer to them as *horizontal contracts*. In this dissertation, we use contracts also to formalize and reason about refinement between two different abstraction levels in the PBD process; for this reason, we refer to this type of contracts as vertical contracts. A vertical contract formalizes the interaction between two abstraction levels via the composition of a model and its vertical refinement, even though they are not directly connected, by connecting them indirectly through a mapping. Informally, this kind of composition captures the fact that the actual satisfaction of all the design requirements and viewpoints by a deployment depends on the supporting execution platform, the underlying physical system (lower level of abstraction), and on the way in which system functionalities (higher level of abstraction) are mapped to them. Formally, this composition can be modelled using two alternative methods, by either contract composition or contract conjunction, based on the specific shape of the contracts specifying the two abstraction levels.

1.5.2 Design Methodology

1.5.2.1 An all-encompassing contract-based design flow

As stated before, a major contribution of this thesis is a compositional and hierarchical design methodology for large-scale CPS, which uses horizontal and vertical contracts to provide formal support for the whole design flow. Contracts were originally developed in a software engineering context [143]. More recently, contract frameworks have been shown to be promising for system design [34], in the context of requirement engineering [50], analysis and verification of hardware and software [10, 58], or requirement monitoring during simulation [141]. In Chapter 4, we introduce instead the first contract-based methodology that encompasses the whole design flow, from top-level requirement formalization to a lower-level representation of the design (system architecture and control algorithm), and covers synthesis from requirements, mapping between abstraction levels, and optimization-based design space exploration.

1.5.2.2 Platform-based design methodology with contracts

The methodology in Chapter 4 also provides the first formal framework capable of combining the platform-based design paradigm with contracts, thus generalizing previous efforts carried out in this direction in the context of analog and mixed-signal system design [152]. This generalization is in turn enabled by a few algorithmic innovations supporting the two main steps in the proposed design flow, namely architecture and control design.

1.5.3 Algorithms

1.5.3.1 Optimized selection of cyber-physical system architectures

In Chapter 5, we address the problem of architecture exploration of cyber-physical systems to minimize a cost function while guaranteeing the desired reliability. We propose two novel algorithms that decrease the problem complexity by combining techniques from formal methods and mathematical programming. Finally, we implement the two algorithms in the *Architecture Exploration* (ArchEx) framework, enabling scalable co-design of large systems for cost and fault tolerance.

We cast the architecture selection problem as an integer linear program on a parametrized graph which models the system architecture. Because exact reliability analysis of such a graph is NP-hard [130], or non-compositional [108], we develop instead a compositional, approximate reliability analysis method which is able to efficiently generate compact expressions for the system failure probability as a function of the system structure and the component failure probabilities. We prove a theoretical bound on the error induced by the approximate algebra and show that it improves on the accuracy of previously proposed approximation schemes [93]. Furthermore, we show that an approximate reliability constraint for the system failure probability can be encoded using a number of linear constraints and associated decision variables that is polynomial in the size of the graph (number of nodes and types of used components).

The first proposed algorithm, denoted as *Integer-Linear Programming with Approximate Reliability (ILP-AR)*, uses the approximate reliability algebra to "eagerly" generate a single, possibly larger, optimization instance which can be efficiently solved to provide a solution within the theoretical bound. The second algorithm, denoted as *Integer-Linear Programming Modulo Reliability (ILP-MR)*, avoids the expensive generation and solution of a large optimization problem via an iterative approach that "lazily" combines integer linear programming with exact reliability analysis. A sequence of smaller optimization problems are solved to provide candidate architecture configurations, which are then analyzed, and accordingly modified until the desired reliability is achieved. The approximate reliability algebra is used to guide these modifications at each step by deriving additional optimization constraints that can accelerate convergence and reduce the number of iterations.

We believe the algorithms above are concrete instances of two general paradigms in design space exploration, which can be used to also handle quality metrics other than reliability.

1.5.3.2 Optimized design of CPS control strategies

Given a system architecture, in Chapter 6, we further demonstrate two paradigms for systematic, contract-based design of control strategies, which merge optimized mapping methods with pre-existing control design techniques. The first paradigm, denoted as *Reactive Synthesis-Based Optimized Control Mapping (RS-OCM)*, enables the generation of hierarchical and distributed controller architectures by combining reactive synthesis from linear temporal logic contracts with simulation-based design space exploration, including monitor-

ing of signal temporal logic contracts from simulation traces. The second paradigm, denoted as *Programming-Based Optimized Control Mapping (P-OCM)* uses instead a formalization of the design requirements and the plant model in terms of arithmetic constraints over real numbers, and formulates the control problem as an optimization problem that is solved within a receding horizon approach to determine a correct control policy that can also optimize some performance metrics.

When a design is built by optimized mapping on a composition of contracts from a pre-characterized library, several instances of contract refinement checking may be required in an optimization run, which can be very expensive to solve for large systems. We then propose an efficient *library-based contract refinement checking* algorithm that relies on a library of contracts with pre-characterized local refinement relations to break down the refinement checking problem into multiple successive refinement checks, each of smaller scale. As in traditional assume-guarantee proof strategies, we decompose the main verification task into smaller sub-tasks, where an aggregation of components is replaced by a more abstract representation from the library [89]. However, in most cases, finding the appropriate abstraction is an issue, since no general guidelines are available to the verification engineer [64, 91]. We instead propose to guide the abstraction process 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 and the system structural decomposition, we can build abstractions automatically on the fly, as we solve the verification problem by successive refinements.

1.5.4 Applications

In this thesis, we demonstrate, for the first time, the effectiveness of a contract-based approach on real-life examples of industrial relevance, namely the design of aircraft electric power distribution and environment control systems. In the context of the aircraft electrical power system, we show how:

i. Compositional and hierarchical refinement of architectures and control algorithms with correctness guarantees is made possible using vertical contracts (Chapter 7);

ii. Optimal selection of large, industrial-scale architectures can be performed in a few minutes (Chapters 5 and 7);

iii. Design validation of reactive controllers based on linear temporal logic contracts shows up to two orders of magnitude improvement in terms of execution time with respect to conventional techniques (Chapter 6);

iv. In addition to guaranteeing system safety, a novel hierarchical optimal load management scheme, denoted as Holms, relying an an efficient mixed integer-linear program solved within a receding horizon approach (following the P-OCM paradigm), can bring substantial performance improvements with respect to state-of-the-art controllers in terms of percentage of shed loads and number of utilized sources (Chapter 7).

Finally, in Chapter 7, we show that the methodology illustrated in detail on the power system example smoothly generalizes to other case studies, such as an aircraft environment control system.

1.6 Organization

This chapter has provided the motivation for our research, including an overview of the aircraft electric power system challenge problem, which will be used throughout this thesis to illustrate our results. We outlined the challenges in the design and design automation of cyber-physical systems. We summarized our strategy to tackle these challenges and highlighted our main contributions. The remainder of the dissertation is organized as follows.

Chapter 2 provides the background material for the concepts introduced in the thesis. We give preliminary notions on Platform-Based Design, assume-guarantee contracts, and formalisms that can be used to express contracts. Moreover, we offer an overview of languages and tools for cyber-physical system modeling, simulation, formal verification and control synthesis, which can be used to implement the contract algebra and the main steps of our design flow. The survey of the related works in this chapter is based on joint work with Alberto Sangiovanni-Vincentelli, Davide Bresolin, Luca Geretti and Tiziano Villa [151].

Chapter 3 and 4 offer the scaffolding for the rest of the thesis. In Chapter 3, we discuss our theoretical results relating assume-guarantee contracts with interface theories, which lead us to further enrich the contract framework at the heart of the methodology, and expose some key computational implications. Moreover, we introduce the notions of heterogeneous refinement and vertical contracts, which are instrumental to formalize the mapping mechanisms in our methodology. In Chapter 4, we detail the main steps of the methodology in terms of architecture design and control design. For both of these steps, we investigate formalisms and tools to formalize requirements using contracts, develop component and contract libraries for design space exploration, and map higher-level specification contracts into lower-level implementation contracts. We then introduce the prototype *Contract-Based Heterogeneous Analysis and System Exploration* (CHASE) environment as a proof-of-concept implementation of the methodology. CHASE facilitates requirement capturing using a front-end pattern-based contract specification language, encodes them using a back-end temporal logic formalism, and reasons about them by leveraging a set of back-end tools. Some of the results presented in Chapter 3 and Chapter 4 are based on joint work with Antonio Iannopollo, Stavros Tripakis, Alberto Sangiovanni-Vincentelli, and Richard Murray [149, 153].

Chapter 5 and Chapter 6 are all centered, respectively, on the architecture and control design steps. In Chapter 5, after introducing the optimal CPS architecture selection problem, we discuss both the ILP-AR and ILP-MR algorithms for reliable and cost-effective architecture design, and their implementation into the ArchEx framework, jointly developed with Nikunj Bajaj, Micheal Masin, and Alberto Sangiovanni-Vincentelli [23]. In Chapter 6, we detail both the RS-OCM and P-OCM approaches to control design, based on joint work with John Finn, Antonio Iannopollo, Alberto Sangiovanni-Vincentelli, and Mehdi

Maasoumy [148, 135]. Moreover, to speed up the RS-OCM approach, we present a library-based algorithm for efficient contract refinement checking, jointly developed with Antonio Iannopollo, Stavros Tripakis and Alberto Sangiovanni-Vincentelli [100].

Chapter 7 illustrates the application of the overall design flow on the two challenge problems provided by the aircraft electrical power and air management systems. Starting from top-level system contracts, we leverage the ILP-MR algorithm and the RS-OCM approach to design both the system architecture and the primary distribution protocol of an electrical power system, based on the work developed in collaboration with Huan Xu, Necmiye Ozay, John Finn, Alberto Sangiovanni-Vincentelli, Richard Murray, Alexandre Donzé and Sanjit Seshia [154]. We then show how the P-OCM approach can be used to design on optimal load management scheme, implemented in the HOLMS framework, jointly developed with Mehdi Maasoumy, Forrest Iandola, Maryam Kamgarpour, Alberto Sangiovanni-Vincentelli, and Claire Tomlin [134]. Finally, we discuss briefly how the proposed methodology can be deployed to perform design space exploration for an aircraft air management system. We draw some conclusions in Chapter 8, and highlight some future research avenues as emerged from this work.

The methodology proposed in this thesis has been supported by a mixed industrial (IBM and United Technology Corporation)-University (U.C. Berkeley and Caltech) consortium, called industrial Cyber Physical systems (iCyPhy), which has been formed in 2013 with the goal of developing methodologies, models and tools for cyber-physical system design. In addition to the the iCyPhy consortium, this research was supported in part by an IBM Ph.D. Fellowship, by the Gigascale Systems Research Center and the Multiscale Systems Center, two of six research centers funded under the Focus Center Research Program (FCRP), a Semiconductor Research Corporation entity, by the TerraSwarm Research Center, one of six centers supported by the STARnet phase of the FCRP, a Semiconductor Research Corporation program sponsored by MARCO and DARPA, and by the National Science Foundation (NSF), via the project "ExCAPE: Expeditions in Computer Augmented Program Engineering."

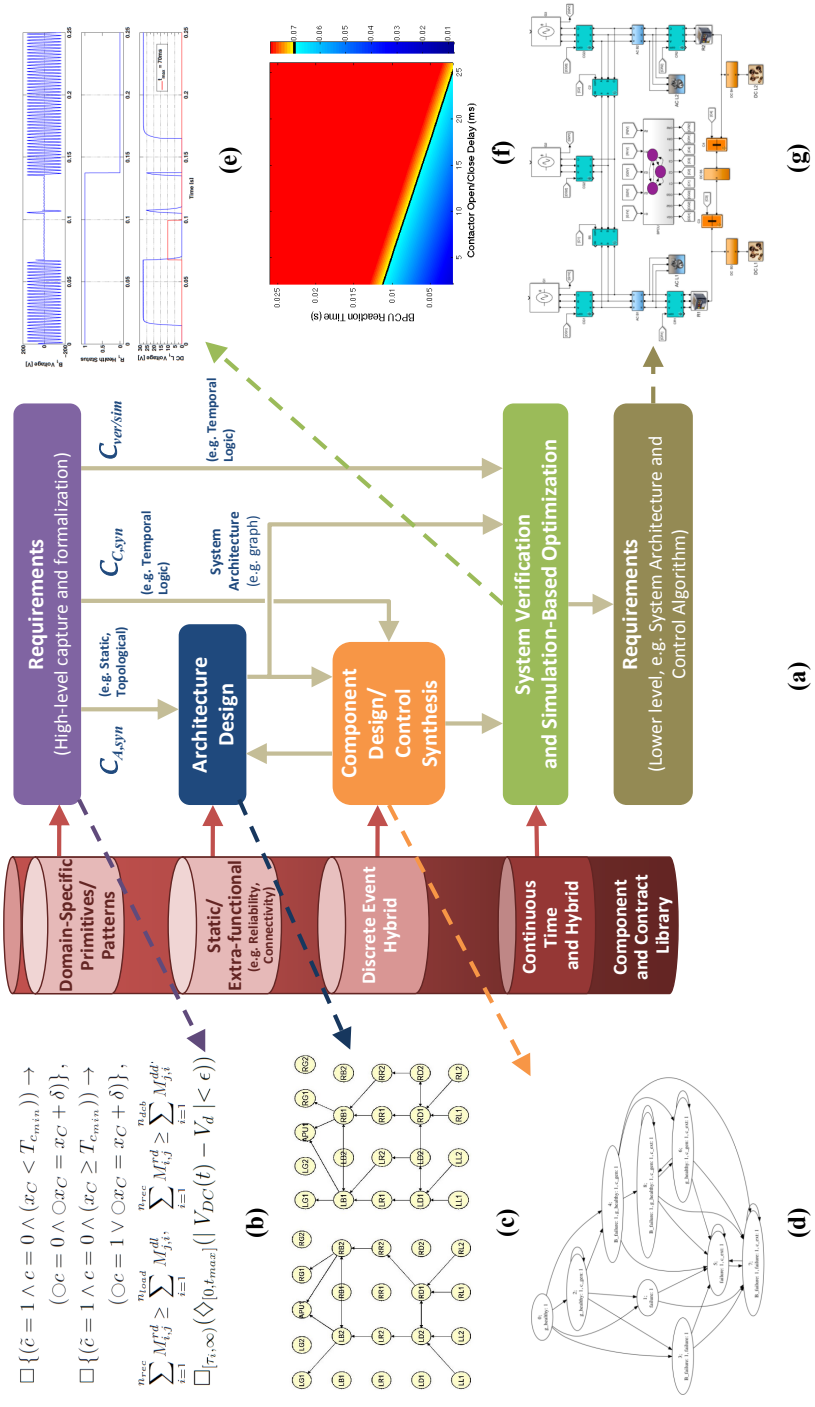


Figure 1.6: (a) Structure of the proposed contract-based methodology for cyber-physical system design, from top-level requirements to the definition of system architecture and control algorithm. Demonstration of the different design steps on the aircraft electric power system example in the dissertation: (b) requirement formalization; (c) architecture selection; (d) reactive control synthesis; (e) simulation-based verification; (f) simulation-based design exploration; (g) hybrid power system model in SIMULINK for further refinement.

Chapter 2

Preliminaries

This chapter provides the background for the concepts introduced in this thesis. We give a preliminary description of Platform-Based Design and contracts. We review formalisms, languages and tools that can be used to specify, analyze or synthesize the design at different levels of abstractions, including the ones of discrete systems and hybrid systems. For each formalism, we expose how the contract operators can be computed, or the challenges in their implementation.

2.1 Platform-Based Design

The pillars for our cyber-physical system design framework are a methodology that relies on the *platform-based design* paradigm [176] and the algebra of *contracts*. *Platform-Based Design* (PBD) was introduced in the late 1980s to capture a design process that could encompass horizontal and vertical decompositions, and multiple viewpoints and, in doing so, support the supply chain as well as multi-layer optimization [176]. In PBD, at each step, top-down refinements of high-level specifications are mapped into bottom-up abstractions and characterizations of potential implementations. Each abstraction layer is defined by a design *platform*, which is the set of all architectures that can be built out of a *library* (collection) of *components* according to *composition rules*. A pictorial representation of a design step in PBD is shown in Figure 2.1.

In the *bottom-up phase* of each design step, we build and model the component library (including both plant and controller). In the *top-down phase*, we formalize the high-level system requirements and we perform an optimization (refinement) phase called *mapping*, where the requirements are mapped into the available implementation library components and their composition. Mapping is cast as an optimization problem where a set of performance metrics and quality factors are optimized over a space constrained by both system requirements and component feasibility constraints. Mapping is the mechanism that allows to move from a level of abstraction to a lower one using the available components within the library. Note that when some constraint cannot be satisfied using the available library

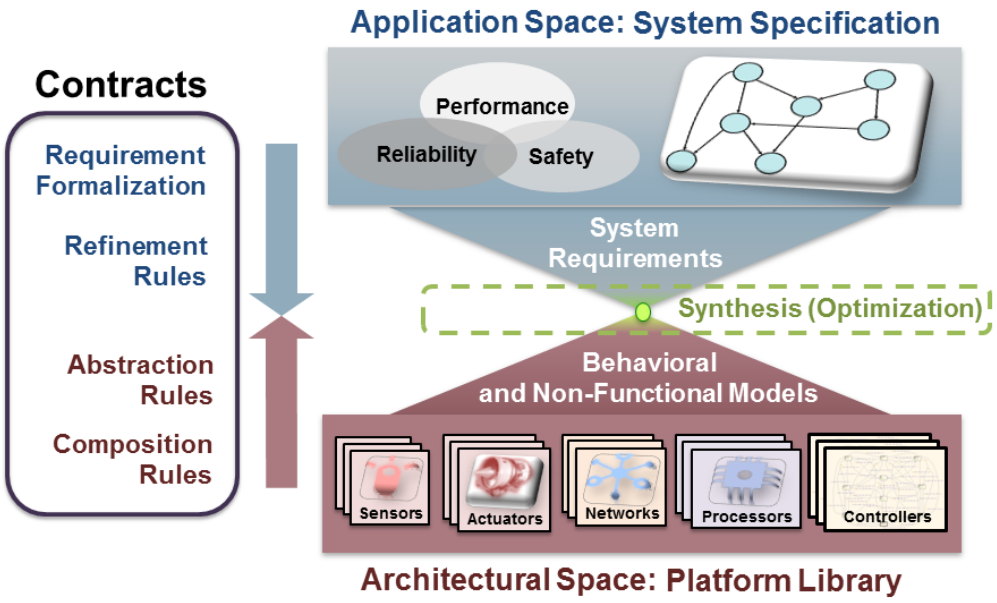


Figure 2.1: Platform-Based Design and the role of contracts.

components or the mapping result is not satisfactory for the designer, additional elements can be designed and inserted into the library. For example, when implementing an algorithm with code running on a processor, we are assigning the functionality of the algorithm to a processor and the code is the result of mapping the "equations" describing the algorithm into the instruction set of the processor. If the processor is too slow, then real-time constraints may be violated. In this case, a new processor has to be found or designed that executes the code fast enough to satisfy the real-time constraint. In the mapping phase, we consider different *viewpoints* (aspects, concerns) of the system (e.g., functional, reliability, safety, timing) and of the components.

If the design process is carried out as a sequence of refinement steps from the most abstract representation of the design platform (top-level requirements) to its most concrete representation (physical implementation), providing guarantees on the correctness of each step becomes essential. Specifically, we seek mechanisms to formally prove that: (i) a set of requirements are *consistent*, i.e. there exists an implementation satisfying all of them; (ii) an aggregation of components are *compatible*, i.e. there exists an environment in which they can correctly operate; (iii) an aggregation of components *refines* a specification, i.e. it implements the specification and is able to operate in any environment admitted by it. Moreover, whenever possible, we require the above proofs to be performed *automatically* and *efficiently*, to tackle the complexity of today's cyber-physical systems (CPS). Therefore,

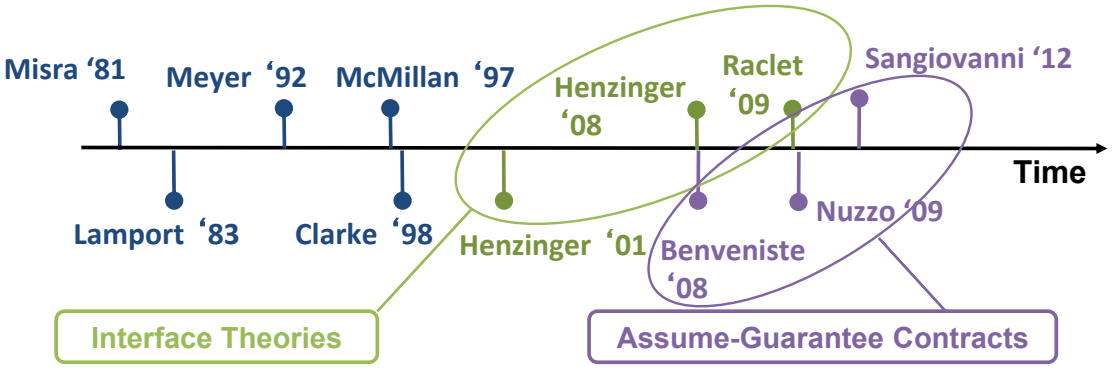


Figure 2.2: Compositional reasoning, contracts, and interfaces in the literature: from assume-guarantee reasoning in formal verification (blue) and contracts in software engineering and object-oriented programming (blue), to interface theories (green) and A/G contracts (purple) for system design. The envelope of contracts has gradually extended from transformational systems to reactive systems over the last decade.

to formalize the above design concepts, and enable the realization of system architectures and control algorithms in a hierarchical and compositional manner that satisfies the constraints and optimizes the cost function(s), we resort to *contracts*.

2.2 Contracts: An Overview

The notion of contracts originates in the context of compositional *assume-guarantee reasoning* [62], which has been used for a long time, mostly for software verification. In a contract framework, design and verification complexity is reduced by decomposing system-level tasks into more manageable subproblems at the component level, under a set of assumptions. System properties can then be inferred or proved based on component properties. Contract frameworks were widely developed in the context of *software engineering* and object oriented programming [143, 6], and further extended in the context of Model Driven Engineering (MDE) [178]. However, their adoption in the context of *reactive systems*, i.e. systems that maintain an ongoing interaction with their environment, as opposed to *transformational system*, considered in object oriented programming, has been advocated only recently [34, 177]. Unlike contracts in software engineering, contracts (and interfaces) for reactive systems need to encompass a richer set of behaviors and models of computation, while extending concepts from *behavioral typing* [29]. As shown in Figure 2.2, different contract theories have been developed over the years, including assume-guarantee contracts [33] and interface theories [70, 78, 168]. However, a thorough investigation of the relations between them is an open research field. Moreover, their concrete adoption in CPS design is still in its infancy, a

major challenge being the absence of a comprehensive modeling formalism for CPS, due to their complexity and heterogeneity [177, 34].

In this thesis, we adopt the *assume-guarantee (A/G) contract* framework, as introduced by Benveniste, et al. [33, 34] to reason about requirements and their refinement during the design process. Because of the explicit distinction between component and environment, A/G contracts are deemed as a rigorous yet *intuitive* framework, which directly conforms to the thought process of a designer, aiming to guarantee certain performance figures for the design under specific assumptions on its environment. Since A/G contracts are centered around *behaviors*, they are *expressive* and versatile enough to encompass all kinds of models encountered in system design, from hardware and software models to representations of physical phenomena [34, 149]. The particular structure of the behaviors is defined by specific instances of the contract model. This will only affect the way operators in the contract algebra are implemented, since the basic definitions will not vary.

An integration language incorporating A/G contracts to formalize system requirements and enable the generation of simulation monitors has been proposed within the META research program [141], with the aim to compress the product development and deployment timeline of defense systems. Furthermore, over the last few years, many publications have demonstrated the application of A/G contracts in different domains, such as automotive [67, 65, 34] and analog integrated systems [152]. In this thesis, we advocate the use of contracts for the entire CPS design flow, including synthesis and optimized mapping of system architectures and control algorithms [154, 100, 135, 153], in addition to system verification. In the following, we start by detailing the notions of components and contracts, and then illustrate the operations and relations of the contract algebra.

2.3 Assume-Guarantee Contracts

Since PBD is based on the composition of components while refining the design, we start our analysis of assume-guarantee contracts with a formal representation of a component and we associate to it a set of properties that the component satisfies expressed with contracts. The contracts will be used to verify the correctness of the composition and of the refinements. A richer notion of a component in the context of PBD will be provided in Chapter 4, as we detail our methodology. In this section, for the sake of conciseness, we refer to the typical, intuitive notion of component as an *open* system, containing some inputs that are provided by other components in the system or the external world, and generating some outputs. We denote the collection of other components and the exterior world as the environment of the component, which is often not completely known when the component is being developed. Components cannot constrain their environment; however, they are designed to be used in a particular context constrained by their contracts.

2.3.1 Components and Contracts

We regard a *component* M as an abstraction representing an element of a design, characterized by a set of (input or output) *variables*, a set of (input or output) *ports*, and a set of *behaviors* over its variables and ports. Components can be connected together by sharing certain ports under constraints on the values of certain variables. In what follows, to simplify, we use the same term variables to denote both component variables and ports. Components can respond to every possible sequence of input variables, i.e. they are receptive to their input variables. Behaviors are generic and could be continuous functions that result from solving differential equations, or sequences of values or events recognized by an automata model. In the following, we also use $[\![M]\!]$ to denote the set of behaviors of a component. A system can then be assembled by *parallel composition* and interconnection of components. We denote the composition of two components M_1 and M_2, when it is defined, as $M_1 \times M_2$. Then, the behaviors of the composition can be described, in general, as the intersection of the behaviors of its components, i.e. $[\![M_1 \times M_2]\!] = [\![M_1]\!] \cap [\![M_2]\!]$. Moreover, a component M may be associated with a contract, offering a *specification* for it.

A *contract* $\mathcal{C}$ for a component M is a triple (V, A, G), where V is the set of component variables, and A and G are assertions, each representing a set of behaviors over V [33]. A represents the *assumptions* that M makes on its environment, and G represents the *guarantees* provided by M under the environment assumptions. A component M satisfies a contract $\mathcal{C}$ whenever M and $\mathcal{C}$ are defined over the same set of variables, and all the behaviors of M satisfy the guarantees of $\mathcal{C}$ in the context of the assumptions, i.e. when $[\![M]\!] \cap A \subseteq G$. We denote this *satisfaction* relation by writing $M \models \mathcal{C}$, and we say that M is an *implementation* of $\mathcal{C}$. However, a component E can also be associated to a contract $\mathcal{C}$ as an *environment*. We say that E is a legal environment of $\mathcal{C}$, and write $E \models_E \mathcal{C}$, whenever E and $\mathcal{C}$ have the same variables and $[\![E]\!] \subseteq A$.

Example 1 (Components and Contracts)**.** *As an example, we consider the amplifier component* Amp *represented in Figure 2.3 (a), whose amplification gain is two. To specify its operation, we can then formulate a simple (stateless) contract as follows:*

$$\mathcal{C}_{amp} = (\{u, y\}, \{(u, y) \in \mathbb{R}^2 |\ |u| \leq 1\}, \{(u, y) \in \mathbb{R}^2 |\ y = 2u\}),$$

where we use $|u|$ to denote the absolute value of u, and constraints (predicates) on the real variables u and y to characterize the sets of assumptions and guarantees of $\mathcal{C}_{amp}$. For brevity's sake, when the domain of all the component variables is known, we can also represent assumptions and guarantees directly in terms of predicates, e.g., $\mathcal{C}_{amp} = (\{u, y\}, |u| \leq 1, y = 2u)$, where we implicitly assume that an assumption predicate ϕ_A and a guarantee predicate ϕ_G are both interpreted over the whole set of contract variables. Moreover, A and G will be, respectively, the set of all the behaviors satisfying ϕ_A and ϕ_G.

The component Amp *duplicates the value of any real number u in the interval $[-1, 1]$, provided as an input. Because the behavior of* Amp *is only determined for a specific input range, there is potentially an infinite number of implementations for $\mathcal{C}_{amp}$. In particular,*

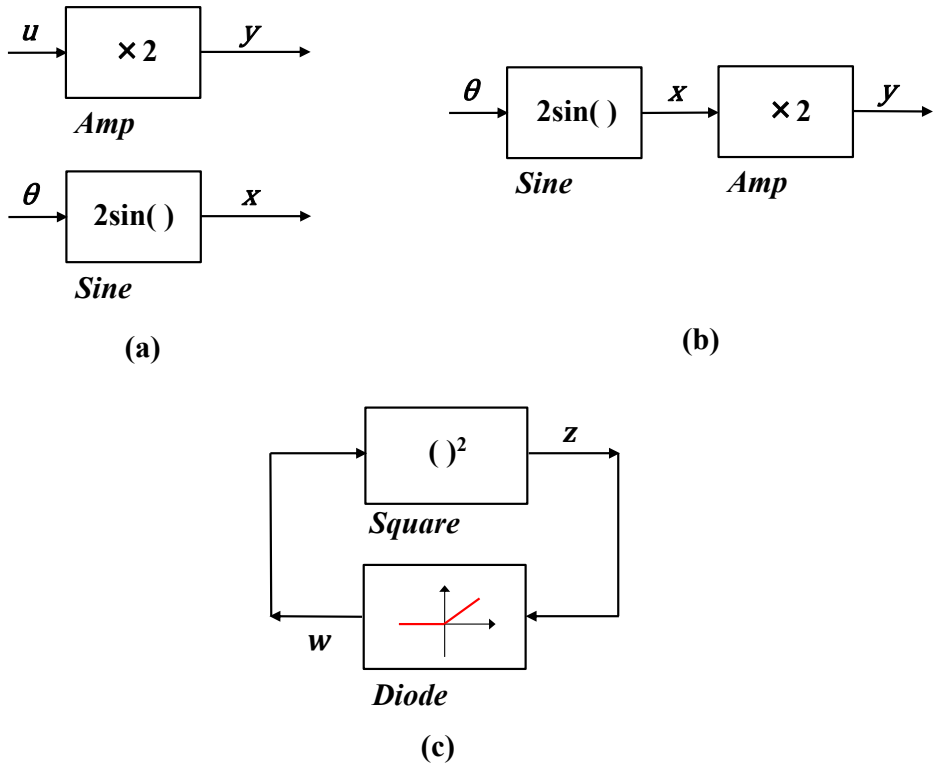


Figure 2.3: Pictorial representation of the components and interconnections used to illustrate some of the contract operations and relations: (a) parallel composition, (b) serial composition, (c) feedback composition.

a component M_{amp}, defined over the same set of variables $\{u, y\}$, and enforcing $y = 2u$ for all $u \in \mathbb{R}$, is certainly an implementation for $\mathcal{C}_{amp}$, i.e. $M_{amp} \models \mathcal{C}_{amp}$. In fact, its set of behaviors $[\![M_{amp}]\!] = \{(u, y) \in \mathbb{R}^2 | y = 2u\}$, coincides with the guarantees of $\mathcal{C}_{amp}$, and therefore $[\![M_{amp}]\!] \cap A_{amp} \subseteq G_{amp}$ trivially holds. On the other hand, let M'_{amp} be an amplifier with saturation, defined over the same set of variables $\{u, y\}$, and characterized by the following behavioral model:

$$M'_{amp} : \begin{cases} y = 2u & \forall u \in \mathbb{R} : -1 \leq u \leq 1, \\ y = -2 & \forall u \in \mathbb{R} : u < -1, \\ y = 2 & \forall u \in \mathbb{R} : u > 1. \end{cases} \tag{2.1}$$

M'_{amp} blocks its output to a constant value when the magnitude of its input exceeds one. However, in the context of the assumptions A_{amp}, it satisfies G_{amp}; therefore, by definition of contract satisfaction, M'_{amp} is also an implementation of $\mathcal{C}_{amp}$.

Any component satisfying the assumptions of $\mathcal{C}_{amp}$ is a legal environment for it; specifi-

cally, a component E_{amp}, defined over $\{u, y\}$, and providing as an output $u = 0$ for all $y \in \mathbb{R}$ is legal, i.e. $E_{amp} \models_E \mathcal{C}_{amp}$. Moreover, given a legal environment E_{amp}, the composition $E_{amp} \times M_{amp}$, for all implementations M_{amp}, generates a closed system.

Two contracts $\mathcal{C}$ and $\mathcal{C}'$ with identical variables, identical assumptions, and such that $G' \cup \overline{A} = G \cup \overline{A}$, where $\overline{A}$ is the complement of A, possess identical sets of environments and implementations. Such two contracts are then *equivalent*. In particular, any contract $\mathcal{C}$ is equivalent to a contract in *saturated (canonical) form* $\mathcal{C}'$, obtained by taking $G' = G \cup \overline{A}$. In what follows, we assume that all contracts are in saturated form.

Example 2 (Saturated Form)**.** *The contract $\mathcal{C}'_{amp} = (\{u, y\}, |u| \leq 1, |u| \leq 1 \rightarrow y = 2u)$ is equivalent to $\mathcal{C}_{amp}$ in Example 1, since it has the same sets of environments and implementations. However, differently than $\mathcal{C}_{amp}$, $\mathcal{C}'_{amp}$ is in saturated form; its set of guarantees $G'_{amp} = (\{u, y\}, |u| > 1 \vee y = 2u)$ is maximal, and coincides with the union of the behaviors of all its implementations.*

2.3.2 Composition

Contracts associated to different components can be combined according to different rules. Similar to parallel composition of components, *parallel composition* ($\otimes$) of contracts can be used to construct composite contracts out of simpler ones. Let $\mathcal{C}_1 = (V, A_1, G_1)$ and $\mathcal{C}_2 = (V, A_2, G_2)$ be contracts (in saturated form) over the same set of variables V. The composite contract $\mathcal{C}_1 \otimes \mathcal{C}_2$ is defined as the triple (V, A, G) where:

$$A = (A_1 \cap A_2) \cup \overline{(G_1 \cap G_2)} \tag{2.2}$$

$$G = G_1 \cap G_2. \tag{2.3}$$

The composite contract must satisfy the guarantees of both, which explains the operation of intersection in (2.3) [34]. Intuitively, the assumptions of the composite contract should also be the conjunction of the assumptions of each contract, since the environment should satisfy all the assumptions. However, in general, part of the assumptions A_1 will be already satisfied by composing $\mathcal{C}_1$ with $\mathcal{C}_2$, acting as a partial environment for $\mathcal{C}_1$. Therefore, G_2 can contribute to relaxing the assumptions A_1 and *vice versa*.

Example 3 (Producer-Consumer System)**.** *Let us consider a simple producer-consumer system, where the producer M_1 is interconnected in series with the consumer M_2, sharing the variable $y \in \mathbb{R}$. Let $\mathcal{C}_1 = (\{y\}, \mathtt{T}, y > 0)$ and $\mathcal{C}_2 = (\{y\}, y > 0, \mathtt{T})$ be the two contracts specifying the behaviors of M_1 and M_2, respectively, both in saturated form. In this example, both assumptions and guarantees are expressed as predicates on y, and* `T` *is the Boolean value* True*. M_1 guarantees that y is a positive number, which coincides with the assumption made by M_2 on its environment. Then, by applying* (2.2) *and* (2.3)*, we obtain $G = (y > 0)$ and $A = (y > 0) \vee (y \leq 0) =$* `T`*, denoting that the composite system is able to operate in any environment, which is intuitive, since the assumptions of M_2 on its environment are relaxed by the guarantees of M_1.*

Specifically, when computing (2.2), we are interested in the maximum set of behaviors A such that $A \cap G_2 \subseteq A_1$ and $A \cap G_1 \subseteq A_2$, where "maximum" refers to the order of sets by inclusion [34]. This is equivalent to finding:

$$\begin{aligned} A &= \max\{A' | A' \subseteq A_1 \cup \overline{G_2}, A' \subseteq A_2 \cup \overline{G_1}\} \\ &= (A_1 \cup \overline{G_2}) \cap (A_2 \cup \overline{G_1}) \\ &= (A_1 \cap A_2) \cup (A_1 \cap \overline{G_1}) \cup (A_2 \cap \overline{G_2}) \cup (\overline{G_1} \cap \overline{G_2}) \\ &= (A_1 \cap A_2) \cup \overline{G_1} \cup \overline{G_2}, \end{aligned} \tag{2.4}$$

which reduces to (2.2). The last equality in (2.4) stems from the fact that $G = G \cup \overline{A}$ holds for a contract $\mathcal{C} = (V, A, G)$ in saturated form. Contract composition preserves saturated form, that is, if $\mathcal{C}_1$ and $\mathcal{C}_2$ are in saturated form, then so is $\mathcal{C}_1 \otimes \mathcal{C}_2$. Moreover, $\otimes$ is associative and commutative and generalizes to an arbitrary number of contracts. We therefore can write $\mathcal{C}_1 \otimes \mathcal{C}_2 \otimes \cdots \otimes \mathcal{C}_n$.

For composition to be defined, contracts need to be over the same set of variables V. If this is not the case, then, before composing the contracts, we must first extend their behaviors to a common set of variables using an inverse projection type of transformation, which we call *alphabet equalization*. Formally, let $\mathcal{C} = (V, A, G)$ be a contract and let $V' \supseteq V$ be the set of variables on which we want to extend $\mathcal{C}$. The *extension* of $\mathcal{C}$ on V' is the new contract $\mathcal{C}' = (V', A', G')$ where A' and G' are sets of behaviors over V', defined by inverse projection of A and G, respectively. In what follows, we freely compose contracts $\mathcal{C}_1$ and $\mathcal{C}_2$ over arbitrary sets of variables V_1, V_2, by implicitly first taking their extensions to $V = V_1 \cup V_2$.

Example 4 (Parallel Composition)**.** *Consider the component* Sine *shown in Figure 2.3 (a), which receives as input an angle θ and produces an output proportional to the sine of θ. We would like to characterize the contract $\mathcal{C}_{sin} \otimes \mathcal{C}'_{amp}$, specifying the parallel composition of* Amp *and* Sine*, where $\mathcal{C}_{sin} = (\{\theta, x\}, \mathtt{T}, x = 2\sin\theta)$. Moreover, we assume that the components interact by sharing their input variables, which we capture by renaming u as θ. Then, to combine correctly the assumptions and guarantees according to* (2.2) *and* (2.3)*, we first need to extend them to the variable set $\{\theta, x, y\}$, thus obtaining*

$$\begin{aligned} \mathcal{C}''_{amp} &= (\{\theta, x, y\}, |\theta| \leq 1, (|\theta| \leq 1) \rightarrow (y = 2\theta)) \\ \mathcal{C}'_{sin} &= (\{\theta, x, y\}, \mathtt{T}, x = 2\sin\theta)\,. \end{aligned}$$

Finally, we can compute the assumptions and guarantees of the composite contract as follows:

$$\begin{aligned} G_\otimes &:= (x = 2\sin\theta) \wedge ((y = 2\theta) \vee (|\theta| > 1)) \\ A_\otimes &:= (|\theta| \leq 1) \vee (x \neq 2\sin\theta) \vee ((y \neq 2\theta) \wedge (|\theta| \leq 1)) \\ &= (|\theta| \leq 1) \vee (x \neq 2\sin\theta). \end{aligned}$$

As informally introduced by the producer-consumer example above, both *serial* and *feedback compositions* of contracts can be defined using the notion of parallel composition. Feedback composition in the context of contracts has also been investigated in a seminal paper by Benvenuti et al. [36]. Let $\mathcal{C}_1 = (V_1, A_1, G_1)$ and $\mathcal{C}_2 = (V_2, A_2, G_2)$ be two contracts, in which the variable sets $V_1 = U_1 \cup Y_1$ and $V_2 = U_2 \cup Y_2$ are, respectively, partitioned into finite sets of input (U_1, U_2) and output variables (Y_1, Y_2). Then, a *serial interconnection structure* σ, defined as a subset of pairs of $Y_1 \times U_2$, i.e. $\sigma \subseteq Y_1 \times U_2$, generates a renaming on $\mathcal{C}_2$ where, for each pair $(y, u) \in \sigma$, u is renamed as y. Let $Y_1^\sigma = \{y | \exists u : (y, u) \in \sigma\}$ and $U_2^\sigma = \{u | \exists y : (y, u) \in \sigma\}$. As represented in Figure 2.4 (a), we can then define a *renaming* operator on $\mathcal{C}_2$, $ren_\sigma(\mathcal{C}_2)$, which returns a new contract $\mathcal{C}_2^\sigma = (U_2 \backslash U_2^\sigma \cup Y_1^\sigma \cup Y_2, A_2^\sigma, G_2^\sigma)$, where A_2^σ and G_2^σ are obtained from A_2 and G_2 after renaming their respective variables according to σ. Finally, we can define the *serial composition* of $\mathcal{C}_1$ and $\mathcal{C}_2$ as $\mathcal{C}_1 \overset{\sigma}{\rightsquigarrow} \mathcal{C}_2 := \mathcal{C}_1 \otimes \mathcal{C}_2^\sigma$.

Example 5 (Serial Composition)**.** *We compute the cascade composition $\mathcal{C}_{sin} \overset{\sigma}{\rightsquigarrow} \mathcal{C}_{amp}$ for $\sigma = \{(x, u)\}$, as shown in Figure 2.3 (b). After renaming and alphabet equalization, by using* (2.2) *and* (2.3)*, we obtain:*

$$G_\sigma := (x = 2\sin\theta) \wedge ((y = 2x) \vee (|x| > 1))$$
$$A_\sigma := (|x| \leq 1) \vee (x \neq 2\sin\theta),$$

where both predicates are now to be interpreted on $\{\theta, x, y\}$.

Similarly, a *feedback interconnection structure* κ can be defined as a subset of pairs $\kappa \subseteq (Y_1 \times U_2) \cup (Y_2 \times U_1)$, thus generating a renaming on both $\mathcal{C}_1$ and $\mathcal{C}_2$ where, for each pair $(y, u) \in \kappa$, u is renamed as y. Let $Y_1^\kappa = \{y \in Y_1 | \exists u \in U_2 : (y, u) \in \kappa\}$, $U_2^\kappa = \{u \in U_2 | \exists y \in Y_1 : (y, u) \in \kappa\}$, $Y_2^\kappa = \{y \in Y_2 | \exists u \in U_1 : (y, u) \in \kappa\}$, and $U_1^\kappa = \{u \in U_1 | \exists y \in Y_2 : (y, u) \in \kappa\}$. We can then define a *renaming* operator ren_κ on $\mathcal{C}_1$ and $\mathcal{C}_2$, , which returns the new contracts $\mathcal{C}_1^\kappa = (U_1 \setminus U_1^\kappa \cup Y_2^\kappa \cup Y_1, A_1^\kappa, G_1^\kappa)$, and $\mathcal{C}_2^\kappa = (U_2 \setminus U_2^\kappa \cup Y_1^\kappa \cup Y_2, A_2^\kappa, G_2^\kappa)$, as represented in Figure 2.4 (b). Finally, we can define the *feedback composition* of $\mathcal{C}_1$ and $\mathcal{C}_2$ as $\mathcal{C}_1 \circ_\kappa \mathcal{C}_2 := \mathcal{C}_1^\kappa \otimes \mathcal{C}_2^\kappa$.

Example 6 (Feedback Composition)**.** *We compute the feedback composition of a* Square *component, which squares any input value, with a* Diode *component, which propagates its input to the output only if it is larger or equal to zero, as shown in Figure 2.3 (c). We assume that the components are formally specified by the contracts $\mathcal{C}_{square} = (\{w, z\}, \mathtt{T}, z = w^2)$ and $\mathcal{C}_{diode} = (\{w, z\}, z \geq 0, (z < 0) \vee (w = z))$. Then, we obtain*

$$G_\kappa := (z = w^2) \wedge (z < 0 \vee w = z) = (z = w^2) \wedge (w = z) = (w = z) \wedge (z = 0 \vee z = 1)$$
$$A_\kappa := (z \geq 0) \vee (z \neq w^2) \vee (z \geq 0 \wedge w \neq z) = \mathtt{T}.$$

A special case of feedback interconnection occurs when a set of outputs of a contract is directly connected to a set of its inputs, as represented in Figure 2.4 (c). For instance, given a contract $\mathcal{C} = (V, A, G)$, in which $V = U \cup Y$, with U and Y finite sets of input and

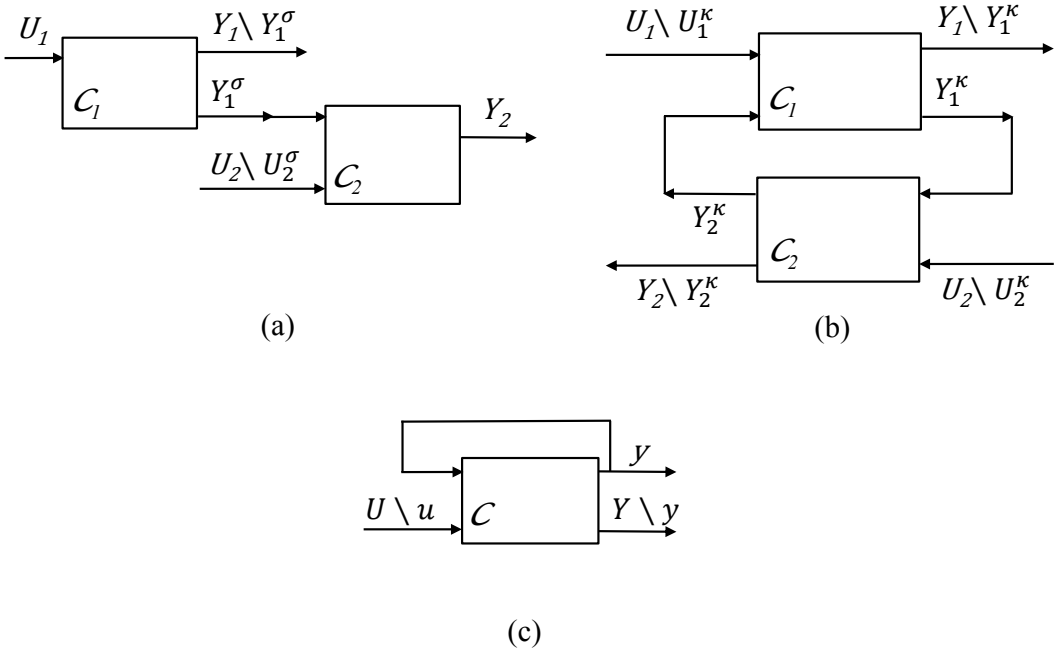


Figure 2.4: Pictorial representation of different examples of contract compositions: (a) serial composition, (b) feedback composition of two contracts, (c) feedback composition of one contract.

output variables, and $U \cap Y = \emptyset$, let $\kappa = (y, u) \in Y \times U$ be a *feedback interconnection* on $\mathcal{C}$, connecting an output of $\mathcal{C}$ to one of its inputs, and let $\mathcal{C}_{id,\kappa} = (\{y, u\}, \mathtt{T}, y = u)$ a contract which guarantees that the variables supposed to be connected in κ are set to be equal. To simplify, we express the guarantees of $\mathcal{C}_{id,\kappa}$ by using a stateless constraint over its variable set; however, stateful extensions, including temporal constructs, are straightforward. We can then reduce the feedback connection on a contract to the general case of feedback composition defined above, by redefining the new contract generated by κ as $\kappa(\mathcal{C}) := \mathcal{C} \circ_\kappa \mathcal{C}_{id,\kappa} = \mathcal{C}^\kappa$.

2.3.3 Compatibility and Consistency

$\mathcal{C}$ is *compatible* if there exists a legal environment E for it, i.e. if and only if $A \neq \emptyset$. The intent is that a component satisfying contract $\mathcal{C}$ can only be used in the context of a compatible environment, to be assured that its behaviors conform with the ones specified by the contract. Similarly, a contract is *consistent* when the set of implementations satisfying it is not empty, i.e. it is feasible to develop implementations for it. For contracts in saturated form, this amounts to verify that $G \neq \emptyset$.

When there is a clear distinction between input (*uncontrolled*) and output (*controlled*) variables, different notions of contract compatibility and consistency can be defined [33, 177,

149]. Let $U \subseteq V$ and $Y \subseteq V$ be, respectively, the subset of input and output variables of $\mathcal{C}$, with $U \cap Y = \emptyset$. Then $\mathcal{C}$ is compatible if and only if A is *Y-receptive*, i.e. if and only if for all behaviors ρ' restricted to variables in Y, there exists a behavior $\rho \in A$, such that ρ' and ρ coincide over Y. Intuitively, an environment has no control on the variables set by an implementation, and therefore A accepts any history offered to the subset Y of its variables. Similarly, $\mathcal{C}$ is consistent if and only if G is *X-receptive*.

Example 7 (Compatibility and Consistency)**.** *Based on these definitions, $\mathcal{C}_{amp}$ and $\mathcal{C}_{sin}$ are both compatible and consistent, while a contract*

$$\mathcal{C}_{amp1} = (\{u, y\}, |u| \leq 2, |u| \leq 1 \rightarrow (y = 2u) \wedge (y > 3))$$

is compatible but inconsistent. In fact, compatibility checking amounts to ask whether $\forall y : |u| \leq 2$ is satisfiable, which is true. On the other hand, consistency checking produces $\forall u : |u| \leq 1 \rightarrow (y = 2u) \wedge (y > 3) =$ `F` (`F` being the Boolean value False*), since it is impossible to satisfy the guarantees of $\mathcal{C}_{amp1}$ for any u in the interval $[-1, 1]$.*

In several practical situations, we are interested in contracts that are compatible and consistent at the same time, i.e. satisfying $G \cap A \neq \emptyset$, to discard pathological situations of contracts which are compatible but not consistent, or consistent but not compatible.

The definitions above can be lifted to pairs of contracts, so that $\mathcal{C}_1$ and $\mathcal{C}_2$ are compatible (consistent) if and only if $\mathcal{C}_1 \otimes \mathcal{C}_2$ is compatible (consistent). As an example, we consider compatibility and consistency conditions for the cascade of contracts in Figure 2.4 (a). To be concrete, we assume that the assumptions and guarantees of $\mathcal{C}_1$ and $\mathcal{C}_2$ are represented in terms of predicates or logic formulas on their variables, i.e. $\mathcal{C}_1 = (U_1 \cup Y_1, \phi_{A1}, \phi_{G1})$ and $\mathcal{C}_2 = (U_2 \setminus U_2^\sigma \cup Y_1^\sigma \cup Y_2, \phi_{A2}, \phi_{G2})$. We can then compute assumptions and guarantees for the composite contract $\mathcal{C}_\sigma = \mathcal{C}_1 \overset{\sigma}{\rightsquigarrow} \mathcal{C}_2$ by applying (2.2) and (2.3) as follows:

$$\phi_{G\sigma} = \phi_{G1} \wedge \phi_{G2} \tag{2.5}$$

$$\phi_{A\sigma} = (\phi_{A1} \wedge \phi_{A2}) \vee \neg\phi_{G1} \vee \neg\phi_{G2}, \tag{2.6}$$

where $\phi_{G\sigma}$ and $\phi_{A\sigma}$ must be interpreted as predicates or formulas over the entire set of variables $U_1 \cup Y_1 \cup U_2 \setminus U_2^\sigma \cup Y_2$. In a general case, we would already conclude that $\mathcal{C}_1$ and $\mathcal{C}_2$ are compatible if and only if $\phi_{A\sigma}$ is satisfiable. Similarly, $\mathcal{C}_1$ and $\mathcal{C}_2$ are consistent if and only if $\phi_{G\sigma}$ is satisfiable. However, this result may not be satisfactory in the special case of contracts with a distinction between controlled and uncontrolled variables, which is still relevant to several application domains, as illustrated by the following examples.

Example 8 (Detecting Incompatibility)**.** *We would like to analyze a system model built by interconnecting two blocks in a modeling environment such as* Simulink[1]*, as shown in Figure 2.5.* L *is a legacy block, seen as a black box, on the behaviors of which we have no information.* Div *produces a real output z which is the inverse of its real input y. The goal for the overall system would be to provide the inverse of any "legal" output of* L*.*

[1] `http://www.mathworks.com/products/simulink`

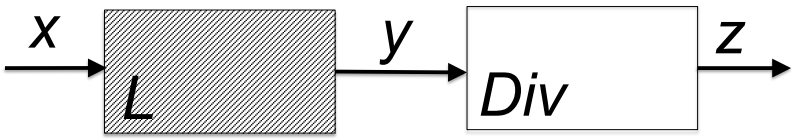


Figure 2.5: Example of a system obtained by assembling a legacy black-box block L and a division component Div.

To specify the behavior of Div*, we can use a simple (static) contract* $\mathcal{C}_D$*, expressed as follows, in terms of assumptions on the environment and guarantees of the component:*

$$\mathcal{C}_D : \begin{cases} \text{variables:} & y, z \in \mathbb{R} \\ \text{assumptions:} & y \neq 0 \\ \text{guarantees:} & z = 1/y \end{cases}, \tag{2.7}$$

where behaviors are specified using constraints on real numbers. On the other hand, a contract $\mathcal{C}_{L1}$ *for* L *would allow any real value both as an input and as an output.*

In such a situation, we would like to conclude that the two specifications (contracts) for L *and* Div *are "incompatible," since there is no way to guarantee that the output of* L *is always a "legal" input for* Div*, i.e. is different than zero. Interestingly, such a concept of incompatibility could be directly detected while computing the composition of the specifications using the theory of relational interfaces [197]. However, this is not necessarily the case in the generic A/G contract framework. In fact, using the formulas reported in Section 2.3.2, it is possible to compute the composition of* $\mathcal{C}_{L1}$ *and* $\mathcal{C}_D$ *as follows:*

$$\mathcal{C}_{L1} \otimes \mathcal{C}_D : \begin{cases} \text{variables:} & x, y, z \in \mathbb{R} \\ \text{assumptions:} & y \neq 0 \\ \text{guarantees:} & (z = 1/y) \vee (y = 0). \end{cases} \tag{2.8}$$

The contract in (2.8) *seems to suggest that any environment capable of enforcing* $y \neq 0$ *would be legal for* $\mathcal{C}_{L1} \otimes \mathcal{C}_D$*. Yet,* y *has now become an "internal" variable for* $\mathcal{C}_{L1} \otimes \mathcal{C}_D$*, and cannot be modified by any environment of the composite contract.*

Example 9 (Inferring Environment Assumptions)**.** *We consider again the system in Figure 2.5. However, we assume now that a more detailed specification is available for the behaviors of* L*, stating that* L *accepts any real input* x *and provides an output* $y > x$*. Such a specification can be expressed using the contract:*

$$\mathcal{C}_{L2} : \begin{cases} \text{variables:} & x, y \in \mathbb{R} \\ \text{assumptions:} & (x, y) \in \mathbb{R}^2 \\ \text{guarantees:} & y > x. \end{cases} \tag{2.9}$$

In this situation, we would like to conclude that $\mathcal{C}_{L2}$ *and* $\mathcal{C}_D$ *are now "compatible." Intuitively, we can observe that any value of* $x \geq 0$ *is guaranteed to provide a nonzero input for* Div*. Hence, there is a way to guarantee that the output of* L *is always "legal" for* Div*. On the other hand, for all* $x < 0$*, there is always a possibility for* L *to violate the assumptions of* Div*, which would make the interconnection "illegal." We would like to automatically derive such a conclusion from the contract composition of* $\mathcal{C}_{L2}$ *and* $\mathcal{C}_D$*. However,* $\mathcal{C}_{L2} \otimes \mathcal{C}_D$ *will be of the form:*

$$\mathcal{C}_{L2} \otimes \mathcal{C}_D : \begin{cases} \text{variables: } x, y, z \in \mathbb{R} \\ \text{assumptions: } y \neq 0 \vee y \leq x \\ \text{guarantees: } y > x \wedge (z = 1/y \vee y = 0) \end{cases},$$

where the desired information about the set of legal environments is still somewhat "hidden" into the assumptions of the composite contract. Again, the desired "compatibility" condition $x \geq 0$ *can be directly obtained out of composition within the theory of relational interfaces.*

As suggested by the examples above, to fully recover some useful features of specification theories for compositional design, such as automatic compatibility checking, some of the basic definitions introduced in this chapter for the A/G contract operators may need to be refined. In Chapter 3, we aim to do so by reconciling the input/output (controlled/uncontrolled) "view" of interface theories with the behavioral approach of A/G contracts. Specifically, we propose a mapping of relational interfaces into A/G contracts, and analyze the preservation properties of key operators and relations (composition, refinement and conjunction) of the two theories under this mapping. Therefore, we defer to Chapter 3 the full treatment of compatibility and consistency checking under composition, including serial and feedback interconnections.

2.3.4 Refinement and Conjunction

Refinement is a preorder on contracts, which formalizes a notion of substitutability. We say that $\mathcal{C}$ refines $\mathcal{C}'$, written $\mathcal{C} \preceq \mathcal{C}'$ (with $\mathcal{C}$ and $\mathcal{C}'$ both in saturated form), if and only if $A \supseteq A'$ and $G \subseteq G'$[2]. Refinement amounts to relaxing assumptions and reinforcing guarantees, therefore strengthening the contract. In other words, contract $\mathcal{C}$ refines $\mathcal{C}'$, if $\mathcal{C}$ admits less implementations than $\mathcal{C}'$, but more legal environments than $\mathcal{C}'$. This is a standard concept inspired by the notion of behavioral subtyping [29]. Clearly, if $M \models \mathcal{C}$ and $\mathcal{C} \preceq \mathcal{C}'$, then $M \models \mathcal{C}'$. On the other hand, if $E \models_E \mathcal{C}'$, then $E \models_E \mathcal{C}$. We can then replace $\mathcal{C}'$ with $\mathcal{C}$.

Example 10 (Refinement)**.** *Let* $\mathcal{C}_{range} = (\{u, y\}, |u| \leq \frac{1}{2}, |y| \leq 1)$ *be a contract specifying the input and output ranges for the component* Amp *in Figure 2.3 (a); we would like to show that* $\mathcal{C}_{amp} \preceq \mathcal{C}_{range}$*, that is, when operating in the context of the assumptions of* $\mathcal{C}_{range}$*,*

[2]It can be useful to recall an equivalent characterization of refinement that also holds for contracts that are not in saturated form. We say that $\mathcal{C} \preceq \mathcal{C}'$, where $\mathcal{C}$ and $\mathcal{C}'$ need not be in saturated form, if and only if $A' \subseteq A$ and $G \cap A' \subseteq G'$.

$\mathcal{C}_{amp}$ produces an output within the range prescribed by the guarantees of $\mathcal{C}_{range}$. To do this, we apply the definitions above to the saturated versions of the two contracts; then, refinement checking translates into proving the validity of the following two predicates involving, respectively, the assumptions and the guarantees of both contracts:

$$|u| \leq 1/2 \rightarrow |u| \leq 1 \tag{2.10}$$
$$(y = 2u) \vee (|u| > 1) \rightarrow (|y| \leq 1) \vee (|u| > 1/2). \tag{2.11}$$

While (2.10) *is trivially true, to show the validity of* (2.11)*, we recall that the antecedent in* (2.11) *is true when either* $(|u| > 1)$ *or* $(y = 2u)$ *holds, and prove that in both cases the consequent is also true. In fact, in the former case, we also have that* $(|u| > 1/2)$ *holds and the implication is true; in the latter case, if* $(1/2 < |u| \leq 1)$ *is true, then the implication is still trivially true. If instead* $(|u| \leq 1/2)$ *is true, we can still conclude* $|y| = 2|u| \leq 1$*, hence* (2.11) *is true.*

Alphabet equalization is also needed as a preliminary step to define refinement when $\mathcal{C}$ and $\mathcal{C}'$ are defined over a different alphabet. A more general case of refinement occurs when $\mathcal{C}$ and $\mathcal{C}'$ are also expressed by using different formalisms, which we denote as *heterogeneous refinement*. In this case, we need to enrich the notion of refinement between contracts via a transformation $\mathcal{M}$ (e.g. a type of projection or inverse projection) that maps the behaviors expressed by one of the contracts to the domain of the other contract, which is generally more involved than alphabet equalization. Heterogeneous refinement is essential in platform-based design flows, and will be discussed in Chapter 3.

To compose multiple requirements on the *same component*, possibly representing different viewpoints that need to be satisfied simultaneously, we can also define the *conjunction* $(\wedge)$ of contracts. Let $\mathcal{C}_1 = (V, A_1, G_1)$ and $\mathcal{C}_2 = (V, A_2, G_2)$ be contracts (in saturated form) over the same set of variables V and on the same component M. We would like to combine $\mathcal{C}_1$ and $\mathcal{C}_2$ into a joint contract $\mathcal{C}_1 \wedge \mathcal{C}_2$ so that, if $M \models \mathcal{C}_1 \wedge \mathcal{C}_2$, then $M \models \mathcal{C}_1$ and $M \models \mathcal{C}_2$. We can compute the conjunction of $\mathcal{C}_1$ and $\mathcal{C}_2$ by taking their greatest lower bound with respect to the refinement relation, i.e. (i) $\mathcal{C}_1 \wedge \mathcal{C}_2$ is guaranteed to refine both $\mathcal{C}_1$ and $\mathcal{C}_2$, and (ii) for any contract $\mathcal{C}'$ such that $\mathcal{C}' \preceq \mathcal{C}_1$ and $\mathcal{C}' \preceq \mathcal{C}_2$, we have $\mathcal{C}' \preceq \mathcal{C}_1 \wedge \mathcal{C}_2$. For contracts in saturated form and on the same alphabet, we have

$$\mathcal{C}_1 \wedge \mathcal{C}_2 = (A_1 \cup A_2, G_1 \cap G_2). \tag{2.12}$$

Example 11 (Conjunction)**.** *Let $\mathcal{C}_{range1}$ and $\mathcal{C}_{range2}$ be two contracts restricting the input and output ranges of an* Amp *component, and defined as follows*

$$\mathcal{C}_{range1} = (\{u, y\}, 0 \leq u \leq 1/2, 0 \leq u \leq 1/2 \rightarrow y \geq u)$$
$$\mathcal{C}_{range2} = (\{u, y\}, 0 \leq u \leq 1, 0 \leq u \leq 1 \rightarrow 0 \leq y \leq 3u),$$

where $u, y \in \mathbb{R}$. *Then, we can compute the conjunction* $\mathcal{C}_{range1} \wedge \mathcal{C}_{range2}$ *as*

$$\begin{aligned} A_\wedge &:= (0 \leq u \leq 1/2) \vee (0 \leq u \leq 1) = 0 \leq u \leq 1 \\ G_\wedge &:= (0 \leq u \leq 1/2 \rightarrow y \geq u) \wedge (0 \leq u \leq 1 \rightarrow 0 \leq y \leq 3u) \\ &= (u \leq y \leq 3u) \vee (u > 1/2 \wedge 0 \leq y \leq 3u) \vee (u < 0) \vee (u > 1). \end{aligned}$$

Since $\mathcal{C}_{amp}$ *admits a larger set of inputs, the whole interval* $[-1, 1]$, *and promises* $y = 2u$ *for* $u \in [0, 1]$, *it clearly refines the conjunction contract, hence it refines both* $\mathcal{C}_{range1}$ *and* $\mathcal{C}_{range2}$. *Therefore, any implementation of* $\mathcal{C}_{amp}$, *such as* M_{amp}, *will also implement both* $\mathcal{C}_{range1}$ *and* $\mathcal{C}_{range2}$.

2.3.5 Summary

Contract-based design is emerging as a unifying compositional paradigm for the specification, design, and verification of large-scale complex systems. Indeed, contract and interface theories promise to offer a scaffolding for the deployment of formal methods in a principled and scalable way [34]. However, in spite of the efforts towards unification, different contract frameworks are currently available, and we lack a clear understanding of the relations between them.

In this dissertation, as mentioned in Section 2.2, we choose the theory of A/G contracts proposed by Benveniste et al. [34] as the underlying framework for the development of our methodology, because of its intuitive nature and its generality. A/G contracts mimic the thought process of a designer, who aims at guaranteeing certain behaviors under specific assumptions on its environment. Since requirements are often naturally seen as assertions (sets of behaviors), A/G contracts seem an adequate framework for use in requirements capture. Moreover, unlike previous interface theories[3], A/G contracts are designed as a generic framework that can be instantiated using different formalisms, and can support a rich composition algebra. Therefore, they also seem adequate to manipulate requirements along the different steps of the design flow, and reason about hierarchies of components (via refinement) and multiple viewpoints (via conjunction).

In spite of their promise of dealing with heterogeneous models, two potential limitations of a generic A/G contract framework emerged from our analysis. First, A/G contracts do not preserve the semantics of composition and compatibility of other interface theories, when applied to "signal-flow" models that make a distinction between uncontrolled inputs and controlled outputs (Section 2.3.3). Second, the basic A/G contract operators are not sufficient to deal with hierarchies of models characterized by heterogeneous architectures, or behaviors expressed in heterogeneous formalisms, possibly capturing richer refinement

[3]Interface theories are prevalently based on automata-based formalisms. In most of the instances, conjunction is also not supported [34], or solely defined under suitable restrictions [197]. Moreover, restrictions are often needed (e.g., the notion of Moore interface) to guarantee compositional refinement under feedback connection [197].

relations, including synthesis and optimization-based methods (Section 2.3.4). We propose appropriate extensions to address these issues in Chapter 3.

Finally, we point out that other versions of A/G contracts were proposed over the years, in addition to the theory considered in this thesis. Another form for A/G contracts supports reasoning about complex component interactions by avoiding using parallel composition of contracts to overcome the problems that certain models have with the effective computation of the operators [167, 88]. Instead, composition is replaced with the concept of *circular reasoning* [17]: when circular reasoning is sound, it is possible to check relations between composite contracts based on their components only, without taking expensive compositions. However, compatibility and conjunction are not addressed in this theory. Finally, Doyen et al. [78] propose an interface model where assumptions on input variables and guarantees on output variables are separated in two different logic formulas. This type of "assume-guarantee interfaces" are less expressive than contracts, since the latter can model relations between input and output variables, which cannot be captured in the former.

2.4 Formalisms for System Specification and Modeling

The behaviors captured by contract assumptions and guarantees can be of different kinds (e.g., discrete or continuous, finite or infinite in length) and they can be concretely represented using different formalisms, e.g., automata, temporal logic, differential equations. In the sequel, we review some of the existing formalisms for the specification and modeling of dynamical systems, which can be used to develop concrete contract frameworks for requirement analysis and manipulation within our methodology. Then, in the next sections, we will review languages and tools associated with these formalisms.

2.4.1 Temporal Logic

Temporal logic is a symbolism for representing and reasoning about the evolution of a system over time. Starting from the '80s [164] it has been successfully applied in formal verification, and a flourishing family of temporal logics has been developed both by academy and industry. Because of its "declarative" flavor, temporal logic seems a "natural" language to formalize high-level requirements in terms of contracts. Moreover, especially for discrete-time, discrete-state system representations, the wealth of results and tools in temporal logic and *model checking* can provide a substantial technological basis for requirement analysis [62].

Classical *discrete-time temporal logics* like linear temporal logic (LTL) and computation tree logic (CTL) [62, 139, 79], originally developed to state requirements of hardware and software electronic systems, can indeed be effectively used to describe discrete event (DE) abstractions of CPS. As an example, in the abstraction offered by LTL, a component can be represented as a set of Boolean variables S_{DE}. Then, the behaviors of a component can be described by the infinite sequences of states of the form $\sigma = s_0 s_1 s_2 \dots$ satisfying an

LTL formula, each state s being a valuation of the Boolean variables in S_{DE}. A sample requirement expressible by LTL is the property "An alert must be eventually resolved," which can be formalized by the formula $\Box(alert \rightarrow \Diamond\, sys\_ok)$, where *alert* and *sys_ok* are Boolean component variables. This formula states that every occurrence of the *alert* event (i.e. when *alert* is asserted), as denoted by the *always* ($\Box$) operator, must *eventually* ($\Diamond$) be followed by an occurrence of a *sys_ok* event.

Discrete-time temporal logics, however, lack the expressiveness needed to capture the continuous aspects of the system in a faithful way. To overcome this limitation, temporal logics have been extended in many ways. A first extension, routinely used in the verification of discrete-time hybrid systems, is to replace Boolean variables with first order atoms, including non-linear arithmetic constraints on real numbers [66]. In this way, LTL can express properties like "If the temperature reaches 90 degrees, then it must eventually decrease below 60," using formulas of the form $\Box(t \geq 90 \rightarrow \Diamond\, t < 60)$, which constrains any state where the temperature t is greater or equal to 90 to be followed by a state where the temperature is below 60.

A second possibility is to add operators to express timing constraints between discrete events. This leads to the development of *real-time temporal logics* such as Metric Temporal Logic (MTL) [112]. For instance, real-time temporal logics can express properties like "An alert must be resolved in 10 seconds," by means of the MTL formula $\Box(alert \rightarrow \Diamond_{[0,10]}\, sys\_ok)$, which forces the *sys_ok* event to occur at most 10 time units after the *alert* event.

Real-time temporal logics have been further extended by providing a continuous notion of time, and by making them capable of expressing properties of continuous quantities. The most relevant language in this family of *continuous-signal logics* is signal temporal logic (STL) [136], which combines first order atoms with timing constraints and is able to express properties like "If the temperature reaches 90 degrees, then it must decrease below 60 in at most 10 seconds." Such a property can be formalized by the formula $\Box(t \geq 90 \rightarrow \Diamond_{[0,10]}\, t < 60)$, which constrains any time instant τ_0 where the temperature t is greater or equal to 90 to be followed by a time instant τ_1 where the temperature is below 60 and such that $\tau_1 - \tau_0 \leq 10$.

More recently, some *logics for hybrid-systems* have been introduced, which can express properties of both the discrete and continuous behaviors of a system. Two relevant members of this class are Hybrid Linear Temporal Logic with Regular Expressions (HRELTL) [61], which extends the LTL with regular expressions (RE), and Differential Dynamic Logic ($\mathsf{d}\mathcal{L}$) [161], which can specify correctness properties for hybrid systems given operationally as hybrid programs. An example of a hybrid property is "If the temperature reaches 90, then an alert is raised," which can be formalized by the HRELTL formula $\Box(t \geq 90 \rightarrow \bigcirc\, alert)$, where $\bigcirc$ is the "next discrete event" operator. On the other hand, the hybrid property "for the state of a train controller *train*, the property $z \leq 100$ always holds true when starting in a state where $v^2 \leq 10$ is true," can be expressed by the $\mathsf{d}\mathcal{L}$ formula $v^2 \leq 10 \rightarrow [train]z \leq 100$, where z and v are the position and the velocity of the train, respectively.

In the remainder of this section, we provide further details about LTL and STL, which

will be widely used in this thesis for the design of reactive controllers, to capture top-level requirements, reason about their correctness, consistency and compatibility, and derive implementations by synthesis and optimized mapping, as discussed in Chapter 4 and 6. We then describe how contracts can be expressed using temporal logic.

2.4.1.1 Linear Temporal Logic

While in platform-based design the "component" is regarded as the fundamental entity of a design, and systems are denoted as interconnections of components, as we describe the basics of LTL, we adhere to the classical terminology, which is historically consolidated [22], and defines design abstractions in terms of "systems."

Definition 2.4.1 (System)**.** *A* system Σ *consists of a set* S *of variables, and a set* $[\![\Sigma]\!]$ *of behaviors over* S*. The domain of* S*, denoted by* $dom(S)$ *or* $\mathcal{S}$*, is the set of valuations of* S*.*

Definition 2.4.2 (Atomic Proposition)**.** *An* atomic proposition *is a statement on system variables that has a unique truth value (*`T` *and* `F`*, denoting respectively the Boolean values* True *or* False*) for a given value* s*. Let* $s \in dom(S)$ *be a state of the system (i.e., a specific valuation of its variables) and* p *be an atomic proposition. Then* $s \models p$ *if* p *is* True *at the state* s*. Otherwise,* $s \not\models p$*.*

LTL also includes Boolean connectors such as negation ($\neg$), disjunction ($\vee$), conjunction ($\wedge$), material implication ($\rightarrow$), and two basic temporal modalities, *next* ($\bigcirc$) and *until* ($\mathcal{U}$). By combining these operators, it is possible to specify a wide range of requirements. Given a set AP of atomic propositions, LTL formulas are formed according to the following grammar:

$$\varphi := \mathtt{T} \mid p \mid \neg\varphi \mid \varphi_1 \wedge \varphi_2 \mid \bigcirc \varphi \mid \varphi_1 \,\mathcal{U}\, \varphi_2$$

where $p \in AP$. Formulas involving other operators, including *eventually* ($\Diamond$) and *always* ($\Box$), can be derived from these basic ones.

LTL formulas over AP are interpreted over infinite sequences of states. In the LTL abstraction, we denote such a sequence as a *behavior* of the system. Let $\sigma = s_0 s_1 s_2 \ldots$ be a behavior and φ be an LTL formula. We say that φ holds at position $i \geq 0$ of σ, written $s_i \models \varphi$, if and only if φ holds for the remainder of the sequence starting at position i. Then, a sequence σ satisfies φ, denoted by $\sigma \models \varphi$, if $s_0 \models \varphi$. Then, a system Σ composed of the variables S is said to satisfy φ, written $\Sigma \models \varphi$, if all sequences in $[\![\Sigma]\!]$ satisfy φ. For the formal semantics of LTL we refer the reader to the book by Baier and Katoen [22].

2.4.1.2 Signal Temporal Logic

LTL allows formal reasoning about temporal behaviors of systems with Boolean, discrete-time signals (variables) or sequences of events. To deal with dense-time real signals and hybrid dynamical models that mix the discrete dynamics of a controller with the continuous dynamics of the plant, several logics were introduced over the years, such as Timed

Propositional Temporal Logic [16] and Metric Temporal Logic [112]. Signal Temporal Logic (STL) [136] has been proposed more recently as a specification language for constraints on real-valued signals in the context of analog and mixed-signal circuits. In this thesis, we use STL to refine LTL system requirements into constraints on physical variables (e.g. voltages and currents) expressed using STL constructs.

For a hybrid dynamical model (system, component), we define a *signal* as a function mapping the time domain $\mathbb{T} = \mathbb{R}_{\geq 0}$ (real numbers larger than or equal to zero) to the reals $\mathbb{R}$. An n-dimensional signal $\boldsymbol{q}$ is then a function from $\mathbb{T}$ to $\mathbb{R}^n$ such that $\forall t \in \mathbb{T}$, $\boldsymbol{q}(t) = (q_1(t), \cdots, q_n(t))$, where $q_i(t)$ is the i-th component of vector $\boldsymbol{q}(t)$. When using STL, we conveniently represent the behaviors of the system's variables over time using multi-dimensional signals or sets of signals. Therefore, we assume that a hybrid system (e.g. implemented in a simulator or described by a set of differential equations) takes as input a signal $\boldsymbol{u}(t)$ and computes an output signal $\boldsymbol{y}(t)$. A collection of signals resulting from a simulation of the system is a *trace*, which can also be viewed as a multi-dimensional signal. A trace $\boldsymbol{s}(t)$ that includes all the system input and output signals can then denote a system *behavior*.

In STL, constraints on real-valued signals, or *predicates*, can be reduced to the form $\mu = g(\boldsymbol{q}) \sim \pi$, where g is a scalar-valued function over the signal $\boldsymbol{q}$[4], $\sim \in \{<, \leq, \geq, >, =, \neq\}$, and π is a real number. As in LTL, temporal formulas are formed using temporal operators, *always*, *eventually* and *until*. However, each temporal operator is indexed by intervals of the form (a, b), $(a, b]$, $[a, b)$, $[a, b]$, (a, ∞), or $[a, \infty)$, where each of a, b is a non-negative real-valued constant. If I is an interval, then an STL formula is written using the following grammar:

$$\varphi := \mathtt{T} \mid \mu \mid \neg\varphi \mid \varphi_1 \wedge \varphi_2 \mid \varphi_1 \,\mathcal{U}_I\, \varphi_2.$$

The *always* and *eventually* operators are defined as special cases of the *until* operator as follows: $\Box_I \varphi \triangleq \neg\Diamond_I \neg\varphi$, $\Diamond_I \varphi \triangleq \mathtt{T}\, \mathcal{U}_I\, \varphi$. When the interval I is omitted, we use the default interval of $[0, +\infty)$.

The semantics of STL formulas are defined informally as follows. The signal $\boldsymbol{q}$ satisfies $\mu = g(\boldsymbol{q}) < 2$ at time t (where $t \geq 0$), written $(\boldsymbol{q}, t) \models \mu$, if $g(\boldsymbol{q}(t)) < 2$. It satisfies $\varphi = \Box_{[0,2)}\ (q > -1)$, written $(\boldsymbol{q}, t) \models \varphi$, if for all time $0 \leq t' < 2$, $q(t + t') > -1$. The signal q_1 satisfies $\varphi = \Diamond_{[1,2)} q_1 > 0.4$ iff there exists a time t such that $1 \leq t < 2$ and $q_1(t) > 0.4$. The two-dimensional signal $\boldsymbol{q} = (q_1, q_2)$ satisfies the formula $\varphi = (q_1 > 10)\ \mathcal{U}_{[2.3,4.5]}\ (q_2 < 1)$ iff there is some time t_0 where $2.3 \leq t_0 \leq 4.5$ and $q_2(t_0) < 1$, and for all times t in $[2.3, t_0)$, $q_1(t)$ is greater than 10. We write $\boldsymbol{q} \models \varphi$ as a shorthand of $(\boldsymbol{q}, 0) \models \varphi$. Formal semantics can be found in the original paper by Maler and Nickovic [136].

Parametric Signal Temporal Logic (PSTL) is an extension of STL introduced by Asarin et al. [19] to define *template formulas* containing unknown parameters. Syntactically speaking, a PSTL formula is an STL formula where numeric constants, either in the constraints given by the predicates μ or in the time intervals of the temporal operators, can be replaced by symbolic parameters. These parameters are divided into two types:

[4]It is common to drop time t in the signal notation.

- A *scale* parameter π is a parameter appearing in predicates of the form $\mu = g(\boldsymbol{q}) \sim \pi$,
- A *time* parameter τ is a parameter appearing in an interval of a temporal operator.

An STL formula is obtained by pairing a PSTL formula with a valuation function that assigns a value to each symbolic parameter. For example, consider the PSTL formula $\varphi(\pi, \tau) = \Box_{[0,\tau]} q > \pi$, with symbolic parameters π (scale) and τ (time). The STL formula $\Box_{[0,10]} q > 1.2$ is an instance of φ obtained with the valuation $w = \{\tau \mapsto 10,\ \pi \mapsto 1.2\}$.

2.4.1.3 Temporal Logic and Contracts

Consistent with the representation of component behaviors, both assumptions A and guarantees G of a contract $\mathcal{C}$ can be specified as temporal logic formulas φ_A and φ_G, respectively. In this case, a component M satisfies the contract $\mathcal{C}$ if it satisfies the logical implication $\varphi_A \rightarrow \varphi_G$, while it is a legal environment for $\mathcal{C}$ if it satisfies the formula φ_A. Contract satisfaction can thus be reduced to two specific instances of model checking [62]. Composition and conjunction of contracts $\mathcal{C}_1$ and $\mathcal{C}_2$ can be represented by appropriate Boolean combination of the formulas φ_{A1}, φ_{A2}, φ_{G1} and φ_{G2}. Other operations on contracts, as defined in Section 4.3.1, can be reduced to special instances of the validity or satisfiability checking problem for temporal logic (or quantified temporal logic, as discussed in Chapter 3) as follows:

- In its simplest formulation, *compatibility and consistency* can be checked by testing whether φ_A or φ_G are *satisfiable*. More complex instances of the problem, which rule out contracts that are "trivially" compatible or consistent, can be solved by *vacuity checking* [115];
- *Refinement* is an instance of *validity checking*: $\mathcal{C}_1 \preceq \mathcal{C}_2$ if and only if $\varphi_{A1} \rightarrow \varphi_{A2}$ and $\varphi_{G2} \rightarrow \varphi_{G1}$ are valid formulas (i.e., tautologies for the language).

A solution of the above problems for HRELTL, based on SMT techniques, has been proposed by Cimatti et al. [61]. Contracts expressed as temporal logic formulas have recently appeared in the literature to instantiate a concrete contract framework and proof system for compositional system verification [58].

2.4.1.4 LTL A/G Contracts

As mentioned earlier, LTL is a widespread formalism to reason about reactive systems and perform analysis and synthesis of embedded control software [164, 160, 208]. It may then be useful to concretely express the sets of behaviors A and G of a contract as formulas in LTL. In what follows, we briefly recall how the main A/G contract operators can be mapped into entailment of LTL formulas.

An LTL A/G contract can be seen as a triple $(V, \varphi_A, \varphi_G)$, where φ_A and φ_G are LTL formulas over the set of variables V. For instance, if $V = \{x, y\}$ and x, y are both integer

variables, a possible LTL A/G contract is $(V, \Box x \geq 0, \Box y \geq 0)$. Indeed, an LTL formula can be used to represent a set of behaviors. For example, the formula $\Box x \geq 0$ represents the set of all behaviors where x is never negative.

Most operations on contracts can then be implemented as operations on LTL formulas in a straightforward way. Saturation of $(V, \varphi_A, \varphi_G)$ can be achieved by setting $\varphi_G := \varphi_A \rightarrow \varphi_G$. The parallel *composition* of contracts $\mathcal{C}_1 = (V, \varphi_{A1}, \varphi_{G1})$ and $\mathcal{C}_2 = (V, \varphi_{A2}, \varphi_{G2})$ can be directly defined in terms of LTL formulas as

$$\mathcal{C}_1 \otimes \mathcal{C}_2 = (V, (\varphi_{A1} \wedge \varphi_{A2}) \vee \neg(\varphi_{G1} \wedge \varphi_{G2}), \varphi_{G1} \wedge \varphi_{G2}).$$

We say that contract $\mathcal{C}_1 = (V, \varphi_{A1}, \varphi_{G1})$ *refines* contract $\mathcal{C}_2 = (V, \varphi_{A2}, \varphi_{G2})$ if formulas $\varphi_{A2} \rightarrow \varphi_{A1}$ and $\varphi_{G1} \rightarrow \varphi_{G2}$ are both valid, or equivalently, if $\neg(\varphi_{A2} \rightarrow \varphi_{A1})$ and $\neg(\varphi_{G1} \rightarrow \varphi_{G2})$ are both unsatisfiable. Similarly, compatibility and consistency checking, in their simplest forms, may be reduced to LTL satisfiability problems. Finally, the *conjunction* of two contracts $\mathcal{C}_1 = (V, \varphi_{A1}, \varphi_{G1})$ and $C_2 = (V, \varphi_{A2}, \varphi_{G2})$ can be obtained as

$$C_1 \wedge C_2 = (V, \varphi_{A1} \vee \varphi_{A2}, \varphi_{G1} \wedge \varphi_{G2}).$$

2.4.2 Hybrid Automata

Formalisms following an "imperative" style, such as hybrid automata, can be used to specify functional requirements especially for system portions of limited complexity. For example, describing the intended behavior of a controlled continuous system together with its discrete controller. Then, one can verify the intended behavior versus generic properties such as *safety*, which requires the automata to stay away from a set of "bad" states, as well as verify whether an implementation is a refinement of the hybrid automaton.

Intuitively, a hybrid automaton is a "finite-state automaton" with continuous variables that evolve according to dynamics specified at each discrete *location* (or *mode*). The evolution of a hybrid automaton alternates *continuous* and *discrete* steps. In a continuous step, the location (i.e., the discrete state) does not change, while the continuous variables change following the continuous dynamics of the location. A discrete evolution step consists of the activation of a discrete transition that can change both the current location and the value of the state variables, in accordance with the reset function associated to the transition. The interleaving of continuous and discrete evolutions is decided by the invariant of the location, which must be true for the continuous evolution to proceed, and by the guard predicate of the transition, which must be true for a discrete transition to be active.

For example, the hybrid automaton in Figure 2.6 can be used to specify the required behaviors of a triangle wave generator with period T and amplitude A. In the *Up* mode, the output y of the generator is required to increase with a constant slope until the internal variable t, initially set to zero, and increasing with a slope of one, reaches $\frac{T}{2}$. The generator will then switch to the *Down* mode, where y is required to decrease with the same slope, while t will keep on increasing until it crosses T. Once this threshold is crossed, the generator commutes back to the *Up* mode, while t is reset to zero.

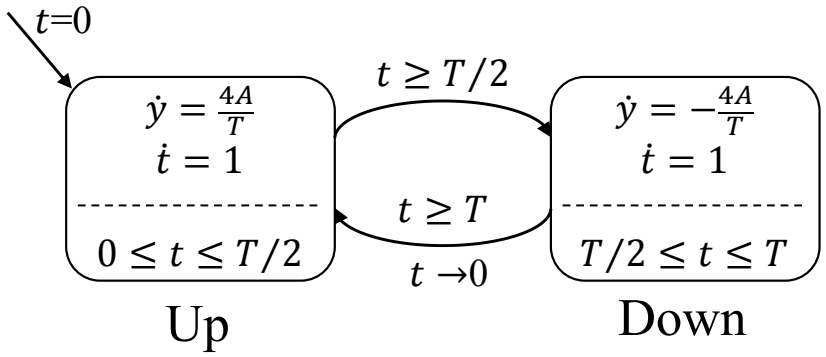


Figure 2.6: Hybrid automaton specifying a triangle wave generator.

Verifying safety of a hybrid automaton with respect to a prescribed set of bad states is equivalent to verifying that all legal behaviors of the automaton do not go through any of the bad states, i.e., the bad states are *unreachable*. The computation of the *reachable set*, which consists of all the states that can be reached under the dynamical evolution starting from a given initial state set, is non-trivial for hybrid automata. Since the states of a hybrid automaton are pairs made by a discrete location together with a vector of continuous variables, they have the cardinality of continuum. Therefore, in general, it is not possible to perform exact reachability analysis.

Hybrid automata come in several flavors. The original model allows for arbitrarily complex dynamics and was developed primarily for algorithmic analysis of hybrid systems [11]. The class of *hybrid input/output automata* enables compositional analysis of systems [132]. In *timed automata* [14] all the continuous variables are *clocks* (they have derivative 1) that can only be reset to zero. Many verification problems are decidable for this class, making it an interesting formalism for verification and requirement analysis. *Rectangular automata* [99] extend timed automata by allowing piecewise constant dynamics, while still keeping decidability of the reachability problem. *Linear hybrid automata* [94] extend rectangular automata by allowing guards and resets to be general linear predicates, at the price of losing decidability.

2.4.2.1 Hybrid Automata and Contracts

We can express contracts with hybrid automata by following the approach proposed by Benvenuti et al. [38]. We model the assumptions A with a hybrid automaton that generates all the admissible input sequences for a component (*uniform assumptions*), while we model the

guarantees G as the set of admissible output sequences for the component. Then, a component M satisfies the contract if the behaviors of the composition of the hybrid automata for A and M are contained in G. When the guarantees are limited to *safety guarantees* ("nothing bad can happen"), then the contract satisfaction problem can be reduced to *reachability analysis* of a composition of automata.

Composition of contracts can be represented by appropriate composition operators on automata. For instance, the conjunction of assumptions corresponds to intersection of the associated automata, while their disjunction can be expressed by non-deterministic choice.

Under suitable restrictions, the other operations on contracts, as defined in Section 4.3.1, can also be reduced to special instances of the reachability problem for timed or hybrid automata. Indeed, compatibility and consistency can be solved by checking whether the set of behaviors of the automaton describing, respectively, A and G is *empty*.

Checking refinement between two contracts $\mathcal{C}_1$ and $\mathcal{C}_2$ is more involved. For uniform assumptions and safety guarantees [38], it is possible to associate to each contract the automaton $\mathcal{H}_A \parallel \mathcal{H}_G$, obtained by composition ($\parallel$) of the two automata $\mathcal{H}_A$ and $\mathcal{H}_G$, respectively describing the contract assumptions and guarantees. $\mathcal{H}_A \parallel \mathcal{H}_G$ models the behaviors admitted by the contract in the context of its legal environments. Then, if $A_2 \subseteq A_1$, contract refinement can be verified by checking the inclusion of the reachable sets of the two hybrid automata $\mathcal{H}_{A_1} \parallel \mathcal{H}_{G_1}$ and $\mathcal{H}_{A_2} \parallel \mathcal{H}_{G_2}$ associated with the contracts. When the evolution of the two hybrid automata cannot be computed exactly, this becomes a difficult task, since it requires computing both over-approximations and under-approximations of the evolution, a capability supported by very few tools.

2.5 Languages and Tools for System Modeling and Simulation

A number of *modeling and interchange languages* have been proposed over the years to enable checking system properties, exploring alternative architectural solutions for the same set of requirements, and exchanging the system descriptions between the different tasks of the design flow (e.g. controller design, validation, verification, testing, and code generation). An exhaustive survey is out of the scope of this dissertation. Among the several languages and tools, we recall here:

- Generic modeling and simulation frameworks, such as MATLAB/SIMULINK[5] and PTOLEMY II[6];
- Hardware description languages, such as Verilog[7], VHDL[8], or transaction-level mod-

[5] http://www.mathworks.com/products/simulink
[6] http://ptolemy.eecs.berkeley.edu
[7] http://www.verilog.com/
[8] http://www.vhdl.org

eling languages, such as SystemC[9], together with their respective analog and mixed-signal extensions[10];

- Modeling languages specifically tailored for acausal multi-physics systems, such as Modelica[11], supported by tools such as DYMOLA[12] or JMODELICA[13];
- Languages for architecture modeling, such as the Systems Modeling Language (SysML)[14] and the Architecture Analysis & Design Language (AADL)[15].

While some of these languages and tools mostly focus on simulation, some others are also geared towards modeling, analysis and verification of extra-functional properties.

A number of proposals have also appeared towards modeling languages specifically tailored to CPS. One of the first examples of these languages is Charon [15]. Charon supports the hierarchical description of system architectures via the operations of instantiation, hiding, and parallel composition. Continuous behaviors can be specified using differential as well as algebraic constraints, all of which can be declared at various levels of the hierarchy. A few years later, Giotto [97] provided an abstract programming model for the implementation of embedded control systems with real-time constraints. Giotto allows the designer to specify time-triggered sensor readings, task invocations, actuator updates, and mode switches in a way that is independent from the implementation details. The code can then be annotated with platform-dependent constraints to automatize the validation of the model and the synthesis of the control software. A more recent modeling language proposal is the Hierarchical Timing Language (HTL) [85]. In HTL critical timing constraints are specified within the language, and forced by the compiler. Programs in HTL are extensible by adding new program modules, and by refining individual program tasks. This mechanism is invariant under parallel composition, and allows individual tasks to be implemented using external languages to ease interoperability.

All the above languages are not intended to be interchange formats, in that they generally lack the capability to easily interface with other tools. A first proposal for a truly platform-independent interchange format based on hybrid automata is the Hybrid System Interchange Format (HSIF) [145]. HSIF can represent networks of hybrid automata, albeit without hierarchy or modules. Variables can be shared or local, and the communication mechanism is based on broadcasting of Boolean signals. Other examples are the METROPOLIS meta-model [159], which also accounts for implementation considerations, such as equation sorting

[9] `http://www.accellera.org/downloads/standards/systemc`

[10] `http://www.eda.org/verilog-ams/`, `http://www.eda.org/vhdl-ams/`, `http://www.systemc-ams.org/`

[11] `https://www.modelica.org/`

[12] `www.dynasim.se/`

[13] `http://www.jmodelica.org/`

[14] SysML is an object oriented modeling language largely based on the Unified Modeling Language (UML) 2.1, which also provides useful extensions for systems engineering (`http://www.omg.org/spec/SysML`).

[15] `http://www.aadl.info/aadl/currentsite`

and event detection, and the interchange format for switched linear systems defined by Di Cairano et al. [74]. More recently, the Compositional Interchange Format (CIF) has been proposed to overcome some of the limitations of previous languages [8], such as the absence of hierarchy in HSIF, and the limitation to linear dynamics only [74]. CIF is a generic exchange format, integrating compositional semantics with automata, process communication and synchronization based on shared events, differential algebraic equations, different forms of urgency, and process definition and instantiation to support re-use and large scale system modeling. It can interface with a number of other languages and tools (e.g. UPPAAL [31], PHAVER [81], ARIADNE [37], MODELICA, MATLAB), and is currently used in both academia and industry.

As an alternative approach to facilitate the integration of different domains and models within a unifying framework, Shah et al. [184] propose the customization of SysML [3] by using profiles and domain specific languages to support multiple representations (or architectures) of the system, and graph transformations to describe the relations between them.

Finally, particularly appealing for CPS modeling and simulation is the Functional Mockup Interface (FMI), an evolving standard for composing component models, which are better realized and characterized using distinct modeling tools [41, 146]. Initially developed within the MODELISAR project, and currently supported by a number of industrial partners and tools[16], FMI shows promise for enabling the exchange and interoperation of model components. The FMI standard supports both co-simulation, where a component called FMU (Functional Mock-up Unit) implements its own simulation algorithm, and model exchange, where an FMU exports sufficient information for an external simulation algorithm to execute simulation. However, while in principle FMI is capable of composing components representing timed behaviors, including physical dynamics and discrete events, several aspects of the standard, e.g. to guarantee that a composite model does not exhibit non-deterministic and unexpected behaviors, are currently object of investigation [48].

2.6 System Verification

As shown in Section 2.4.1 and Section 2.4.2, the operations and relations on temporal logic and hybrid automata contracts can be reduced to basic verification tasks. In this section, we discuss some of the approaches reported in the literature to perform these tasks, together with the tools embodying them. Specifically, we focus on formal verification of hybrid models, which generates, in general, intractable problems, and classify the verification tools into five categories, based on the strategies adopted to deal with intractability.

2.6.1 Exact Reachability Set Computation

When the system dynamics are simple enough to be captured by timed or rectangular automata, their evolution can be computed exactly, and most of the verification techniques for

[16] `https://www.fmi-standard.org/`

finite-state models can be used to obtain an exact answer to verification problems.

A seminal tool in this category is KRONOS [212], which verifies real-time systems modeled by timed automata with respect to requirements expressed in the real-time logic TCTL (Timed Computation Tree Logic), using a backward-forward analysis approach.

The same approach was then extended to support rectangular automata in HyTech [98], by dealing with polyhedral state sets. A key feature of HyTech is its ability to perform parametric analysis, that is, to determine the values of design parameters for which a rectangular hybrid automaton satisfies a temporal-logic requirement. It can then be used as an evaluation engine for optimization-based design exploration, as discussed in Section 4.4.

Modern tools use a different approach, based on an on-the-fly verification algorithm that does not need to build the entire reached set of the system. The most relevant tool using this approach is Uppaal [31], written in Java and C++, and equipped with a graphical user interface. It handles real-time systems modeled as networks of timed automata, and complex properties expressed in a subset of CTL. Since the dynamics are represented just by clocks, it can support models with up to 100 of them. A comparison of the performance of the three tools above on the well-known railroad crossing example can be found in the literature [39].

The above tools are mostly based on a compact, symbolic representation for the infinite component of the state space of a timed automaton (over clock variables), and explicit representations for the finite component (over Boolean state variables). An alternative approach to the verification of timed automata is, instead, based on *fully symbolic* methods, which employ a single symbolic representation for both finite and infinite components of the state space. Examples of fully symbolic model checkers include Red [200] and TMV [180], which was demonstrated on the verification of timed circuits, and is based on a Boolean encoding of difference logic (DL) formulas together with efficient decision procedures for first-order logics involving arithmetic.

2.6.2 Reachable Set Approximations

When the dynamics is more complex, the reachable set cannot be computed exactly. Nevertheless, approximation techniques can be used to obtain an answer in some cases. This approach is mainly used to verify *safety properties*: the system is safe if the reachable set is included in the safe set of states. Hence, *over-approximations* may be used to obtain positive answers, while *under-approximations* give negative answers.

One of the first tools that enabled verification of hybrid systems with complex dynamics is *d/dt* [20]. The tool approximates reachable states for hybrid automata where the continuous dynamics is defined by linear differential equations. Being one of the first approaches, the tool does not allow the composition of automata, and is limited in scalability.

PHAVer [81] handles affine dynamics and guards and supports the composition of hybrid automata. The state space is represented using polytopes. Results are formally sound because of the exact and robust arithmetic with unlimited precision. Scalability is, however,

limited: models with more than 10 continuous variables are usually out of the capabilities of the tool.

SpaceEx [82] improves upon PHAVer in terms of scalability: models with 100 variables have been analyzed with this tool. It combines polyhedra and support functions to represent the state space of systems with piecewise affine, non-deterministic dynamics. Differently from PHAVer, the result of SpaceEx is not guaranteed to be numerically sound. This means that when the tool states that the system is safe, we can only conclude that more sophisticated methods are necessary to find bugs for that system.

Flow* [56] supports systems with non-linear ODEs (polynomial dynamics inside modes, polyhedral guards on discrete transitions) by representing the state space using Taylor models (bounded degree polynomials over the initial conditions and time, bloated by an interval). Results are guaranteed to be numerically sound but scalability is limited to a dozen variables.

Ariadne [37, 38] uses numerical methods based on the theory of computable analysis to manipulate real numbers, functions, and sets in the Euclidean space, in order to verify hybrid systems with non-linear dynamics, guards, and reset functions. It supports composition to build complex systems from simpler components, and can compute both upper-approximations and lower-approximations of the reachable set, which play the role of over and under approximations. By combining them, Ariadne can provide both positive and negative answers to the verification of safety properties and other more complex problems. Its expressivity, however, affects performance and scalability, which is currently limited to models with up to 10 continuous variables.

An alternative approach to approximate the reachable set of a hybrid automaton is to drop the standard infinite precision semantics, and adopt an ϵ-semantics where states whose distance is less than a fixed ϵ are indistinguishable. Under this assumption the reachability problem for hybrid automata becomes decidable [52]. pyHybridAnalysis [51] is a Python package that implements the ϵ-semantics approach to symbolically compute an approximation of the reachability region of hybrid automata with semi-algebraic dynamics.

2.6.3 Discrete Abstractions

In this setting, the hybrid model under verification is first abstracted by a finite-state discrete model that approximates the original one. If the abstraction is not accurate enough to obtain an answer to the verification problem, it is improved until either an answer is found or the maximum number of refinement steps is reached [13, 63]. The main advantage of this approach is that, in some cases, an answer to the verification problem can be obtained with few refinement steps, even for very complex models.

The refinement algorithm proposed Clarke et al. [63] has been implemented by CheckMate [188], a Matlab/Simulink toolbox for the simulation and verification of hybrid systems with linear and affine dynamics. The abstraction of the system is obtained with a method called flow pipe approximation, where the reachable set over a bounded time interval $[0, t]$ is approximated by the union of a sequence of convex polyhedra.

One of the first tools to extend this approach to non-linear systems is HSOLVER [173], which uses constraint propagation and abstraction-refinement techniques to discretize the state space of the system and verify safety properties. HSOLVER supports systems with complex non-linear dynamics and guards, but it does not support the composition of automata. Because of the particular state-space representation, it cannot provide a graphical output of the reachable set, but only a safe/possibly-unsafe answer to the verification problem.

HybridSAL [194] uses predicate abstraction to abstract the discrete dynamics and qualitative reasoning to abstract the continuous dynamics of polynomial hybrid systems. The algorithm can be applied compositionally to abstract a system described as a composition of automata. Results are guaranteed to be sound. Its scalability is limited: only 10 continuous variables can be handled.

HyCOMP [60] uses a different approach, where the system is abstracted with a discrete but infinite-state model using an SMT approach. The abstraction is precise for piecewise constant dynamics and is an over-approximation for affine dynamics. Results are guaranteed to be sound (the SMT-solver uses infinite-precision arithmetic). The tool was tested successfully on models with 60 continuous variables with piecewise constant dynamics and 150 Boolean variables.

2.6.4 Automated Theorem Proving

Given a sufficiently expressive logic, the verification problem can be reduced to test whether a formula of the form $Sys \rightarrow Prop$ is *valid* (a logical tautology), where Sys is a representation of the system under verification and $Prop$ is the property of interest. Automated theorem proving techniques can thus be used to solve the problem. While in principle this approach can easily manage parametric and partially specified systems, and properties of arbitrary complexity, very few tools exploit it in the context of hybrid systems. This is mainly due to the need for a complex temporal logic to describe the system in detail, and to the fact that automated theorem provers usually need some intervention from the user to guide the proof search and find an answer.

A tool using theorem proving techniques in the context of hybrid systems is KeyMaera [161], which combines deductive, real algebraic, and computer algebraic prover technologies. Systems and properties are specified using the temporal logic $\mathtt{d}\mathcal{L}$. To automate the verification process, KeyMaera implements automatic proof strategies that decompose the hybrid system specification symbolically. The tool is particularly suitable for verifying parametric hybrid systems and has been used successfully for verifying collision avoidance in case studies from train control to air traffic management.

2.6.5 Simulation

A simulation-based approach can be used to verify black-box models (when the internal dynamics is unknown), or models of more complex systems, since simulation can be made

more computationally feasible. Clearly, simulation is simply a virtual test bench that gives answers as good as the questions that are asked, hence there is no guarantee that the system behaves correctly under all conditions. Simulation-based verification explores the state space of the system by computing a set of trajectories while hoping to cover as much as possible the relevant parts of the state space. If one of the trajectories violates the property, a *counterexample* is found and a negative answer to the verification problem is given. Otherwise, no conclusion can be made on the truth of the property, since simulation cannot cover the entire state space. Similarly, simulation-based verification cannot be used, in general, to certify the satisfaction of a contract, but rather to monitor and detect possible violations.

A first tool based on simulation is BREACH [77], a MATLAB/C++ toolbox for the simulation, verification of temporal logic properties, and reachability analysis of dynamical systems, defined as systems of ordinary differential equations (ODEs) or by external modeling tools such as SIMULINK. It uses systematic simulation to compute an under-approximation of the reachable set based only on a finite (though possibly large) number of simulations. It supports complex properties in STL and parameter synthesis.

S-TALIRO [18] is also a suite of tools for the analysis of continuous and hybrid dynamical systems using linear time temporal logic. Distributed as a MATLAB toolbox, it uses a robustness metric to guide the state space exploration, exploiting randomized testing and stochastic optimization techniques to maximize the chance of finding a counterexample. Similarly to BREACH, it supports complex properties in Metric Temporal Logic and parametric systems.

Finally, System Level Formal Verification (SLFV) [138] can prove system correctness notwithstanding uncontrollable events (such as faults, variation in system parameters, external disturbances) by exhaustively considering all the relevant simulation scenarios.

2.7 Control Synthesis

Control synthesis deals with the problem of mapping (synthesizing) high-level formal requirements and a description of the plant, into a lower-level, correct-by-construction, controller that implements the desired requirements once it is composed with the plant. We review below the main techniques for the synthesis of control algorithms for CPS.

2.7.1 Reactive Synthesis

When requirements are expressed using a discrete-time temporal logic (e.g. LTL or CTL), controller synthesis can be solved using techniques from *reactive synthesis*, which has been an active area of research since the late 1980s, and it is still attracting a considerable attention today [160, 111, 113, 209]. In this case, the specifications are mapped on a DE implementation of the controller, e.g. in terms of a state machine that represents a lower level of abstraction in the design refinement process.

Let E and D be sets of environment (input) and controlled (output) variables, respectively, of a DE controller. Let $s = (e, d) \in \mathcal{E} \times \mathcal{D}$ be its state, and $\mathcal{C}_{LTL}$ an LTL contract of the form $(E \cup D, \varphi_e, \varphi_e \rightarrow \varphi_s)$, where φ_e characterizes the assumptions on the environment and φ_s characterizes the system requirements. Reactive synthesis can then be viewed as a two-player game between an environment that attempts to falsify the specification in $\mathcal{C}_{LTL}$ and a controlled plant that tries to satisfy it. A control strategy is a partial function $f : (s_0 s_1 \dots s_{t-1}, e_t) \mapsto d_t$, which selects the value of the controlled variables based on the state sequence so far and the behavior of the environment so that the (controlled) system satisfies φ_s as long as the environment satisfies φ_e. If such a strategy exists, the specification is said to be *realizable*. For general LTL, the synthesis problem has a doubly exponential complexity. However, a subset of LTL, namely generalized reactivity (1) (GR(1)), generates problems that are polynomial in $|\mathcal{E} \times \mathcal{D}|$, the number of valuations of the variables in E and D [160]. Given a GR(1) specification, there are game solvers and digital design synthesis tools that generate a finite-state automaton that represents the control strategy for the system [165, 209, 106, 43, 42].

When the requirements also involve continuous variables, by "replacing" continuous dynamics by discrete abstractions it is possible to reduce the synthesis problem to a purely discrete one and therefore within the realm of reactive synthesis, or other established DE system control synthesis methods [170, 53], as available for instance in the third revision of the CIF language for supervisory control synthesis [199]. More recently, a synthesis method for discrete-time CPS subject to STL specifications has been proposed based on a model predictive control framework [171, 172], which can be seen as an instance of the programming-based optimized control mapping (P-OCM) paradigm in Chapter 6. The STL specifications are encoded as mixed integer-linear constraints on the system variables of an optimization problem that is solved at each step, following a receding horizon approach. The synthesis problem is addressed in both the cases of deterministic environment (non-reactive setting) [171] and potentially adversarial environment [172], which results into a robust optimization problem, as the ones formulated in Chapter 4 and Chapter 6.

2.7.2 Synthesis by Abstraction

Because of the limited applicability of existing tools to large-scale CPS hybrid models, constructing effective abstractions in a compositional way is key in order to tackle the synthesis problem. Indeed, the notion of approximate bisimulation [86] has been recently introduced to obtain correct and complete abstractions of differential equations that can be used to solve controller design problems. Pessoa [142] is a software toolbox, which exploits approximate bisimulation to implement efficient synthesis algorithms operating over the equivalent finite-state machine models. The resulting controllers are also finite-state and can be readily transformed into code for any desired digital platform. This transformation assigns the finite-state controller operation to a processor, where code is the result of mapping the controller equations into the instruction set of the processor.

Another approach to mapping a controller into a processor is the control software synthesis tool QKS [140]. Given the sampling time of the controller and the precision of the analog-to-digital conversion of state measurements, QKS can compute both the controllable region and an implementation in C code of a controller driving the system into a goal region in finite time.

A library-based compositional synthesis approach that directly conforms to the PBD paradigm has recently been presented to solve high-level motion planning problems for multi-robot systems [174]. The desired behavior of a group of robots is specified using a set of safe LTL properties (top-down step of the flow). The closed-loop behavior of the robots under the action of different lower-level controllers is abstracted using a library of motion primitives, each of which corresponds to a controller that ensures a particular trajectory in a given configuration (bottom-up step of the flow). By relying on these primitives, the mapping problem is then encoded as an SMT problem and solved by using an off-the-shelf SMT solver to efficiently generate control strategies for the robots.

2.7.3 Hybrid Controller Synthesis

Several real-time constraints, mostly related to the physical plant and the hardware implementation of the controller, may require the full expressiveness of continuous and hybrid models. However, solving the controller synthesis problem by directly mapping to these abstractions is a very difficult task [46]. Even in the context of timed automata, where the synthesis problem is known to be solvable in an exact way [137], efficient and practical tools are lacking. One of the few exceptions is UPPAAL-TIGA [30, 4], an extension of UPPAAL that implements on-the-fly algorithms for solving the controller synthesis problem on timed automata with respect to reachability and safety properties expressed using timed computation tree logic.

Most of the algorithms for controller synthesis of hybrid automata subject to a safety specification are based on solving a differential game in which the environment is trying to drive the system into its target set at the same time as avoiding the target set of the controller. A general formulation for this problem can be found in the literature [195, 26]. Examples include the symbolic semi-algorithm to compute the controllable region of a linear hybrid automaton with respect to a safety goal [206], and a procedure to synthesize the maximal safe controller for more general hybrid systems with a lower bound on event separation [26]. One of the few publicly available tools implementing this two-person game approach is PHAVER+ [32], an extension of PHAVER that can automatically synthesize discrete controllers for linear hybrid automata with respect to safety and reachability goals.

A few works have appeared in the literature addressing the problem of synthesizing switching logic for hybrid systems [102, 103]. See, for example, [104] for a discussion of these techniques as well as an extensive literature survey related to hybrid controller synthesis. Two synthesis (mapping) approaches have also been presented that can incorporate finite-precision sensors and actuators as well as the finite response time of the controller [47, 75]. In these works, the synthesis problem is addressed for two sub-classes of hybrid automata,

namely *elastic controllers*, and *lazy linear hybrid automata*, operating in an environment represented by hybrid automata. Elastic controllers are timed automata without invariants and with closed guards [210, 71]. They were introduced together with a parametric semantics for timed controllers called the Almost ASAP semantics, which relaxes the standard idealized ASAP (As Soon As Possible) semantics that cannot be implemented by any physical device no matter how fast it is. The result is that any correct Almost ASAP controller can be implemented by a program on a hardware if this hardware is fast enough. The first paper [47] presents a corresponding automated tool chain that can extract from an elastic controller a correct-by-construction HW/SW implementation described in SystemC. On the other hand, lazy linear hybrid automata [7, 101] are used in the second paper [75] to model the discrete-time behavior of control systems containing finite-precision sensors and actuators interacting with their environment under bounded delays. The result is a methodology and a corresponding tool chain to synthesize an implementable control strategy for lazy linear hybrid automata.

2.8 Conclusions

We offered an outlook over the main components of our methodology: platform-based design and contracts. We argued that A/G contracts are an intuitive, rich, and flexible contract model that can solidify all the steps in our design flow. We thoroughly illustrated the key operations and relations of the generic A/G contract theory, and pointed out possible gaps in terms of (i) supporting compatibility and consistency in models with uncontrolled inputs and controlled outputs, and (ii) formalizing richer refinement relations between layers described by heterogeneous architectures and behavior formalisms. Addressing these gaps is the object of Chapter 3. We then surveyed formalisms, languages and tools that can be used to build model libraries, concretize the contract algebra, and perform contract-based requirement validation, system verification, and synthesis. While some of the existing languages and tools go already a long way towards efficiently handling several design tasks, a big leap towards managing complexity and heterogeneity is only made possible by an all-encompassing framework that is able to construct appropriate abstraction layers and reason about complex hierarchies of verification and synthesis steps by combining the most suitable tools. Developing such a framework is the objective of Chapter 4.

Chapter 3

Assume-Guarantee Contract Framework for Cyber-Physical System Design

In this chapter we propose two additions to the contract theory introduced in Chapter 2, to fully support multi-view and multi-layer design flows with heterogeneous models. We further investigate the notions of contract compatibility and consistency, and introduce a new projection operator to preserve the semantics of compatibility supported by other interface theories, and encompass models that make a distinction between uncontrolled inputs and controlled outputs. We then extend the notion of refinement, by introducing the concepts of heterogeneous refinement and vertical contracts to deal with hierarchies of models characterized by heterogeneous architectures ("structural heterogeneity"), or behaviors expressed in heterogeneous formalisms ("semantic heterogeneity"). The resulting contract framework, at the heart of the methodology in Chapter 4, can encompass a richer set of refinement relations, including synthesis methods and optimized mappings of specification platforms into implementation platforms.

3.1 Introduction

Methodologies such as component-based design [69] and contract-based design [177] (CBD) are emerging as unifying formal compositional paradigms. They support requirement engineering by providing rigorous formalisms to reason about different abstraction levels in system design. Moreover, they offer mechanisms for early detection of integration errors, e.g., by checking compatibility between the components locally, before performing global system verification.

Yet, different formal theories of components and contracts have been proposed in the literature, and there is currently not enough understanding of the relations between them. As an example, we consider the so-called *interface theories* [69], such as *interface automata* [70]

and *relational interfaces* [197], on the one hand, and the *assume-guarantee (A/G) contract* framework [33, 34], on the other hand. Examining the relation between these two frameworks is interesting because, while having the same overall objectives, they are supported by different mathematical formalisms. For instance, in an A/G contract the assumptions made on the environment and the guarantees provided by the system are modeled as separate sets of behaviors, whereas in interface theories the two are "merged" into a single model, called an *interface*.

In addition, interfaces generally rely on the distinction between inputs and outputs. The fact that an interface may not be *input-complete* (i.e., accept any input at any time) is essential and leads to game-theoretic definitions of composition and refinement. On the other hand, A/G contracts capture assumptions and guarantees as sets of behaviors over a common set of variables, in general with no distinction between inputs and outputs (e.g., for composition). These differences result in different definitions of key elements of the theories, such as composition and refinement, possibly leading to unsatisfactory results when either one of the frameworks is applied to certain types of models, as highlighted by Examples 8 and 9 of Section 2.3.3.

As discussed in Section 2.3, a *behavioral framework* such as A/G contracts is, in general, preferable since it is expressive and versatile enough to encompass all kinds of models encountered in system design, from hardware and software models to representations of physical phenomena. A/G contracts specify components in terms of sets of *behaviors* which assign a history of "values" to their variables or ports. Behaviors are generic and abstract; for instance, they could be continuous functions that result from solving differential equations, or sequences of values or events recognized by an automata model [177]. The particular structure of the behaviors is defined by specific instances of the contract model. This will only affect the way operators in the contract algebra are implemented, since the basic definitions will not vary.

A framework centered around behaviors over variables, without *a priori* distinction between inputs and outputs, is certainly suitable to model the majority of physical (e.g. mechanical, electrical, hydraulic or thermal) components, which are generally governed by laws that merely impose relations (rather than functions) among system variables, and where interconnections mean that variables are shared (rather than assigned) among subsystems. However, there are some important situations, in which a *signal flow* approach is more appropriate, e.g. in signal processing, feedback control based on sensor outputs and actuator inputs, and in systems composed of unilateral devices [202]. In these cases, relations between system variables are better viewed in terms of inputs and outputs, and interconnections in terms of output-to-input assignments. Inputs are used to capture the influence of the environment on the system, while outputs are used to capture the influence of the system on the environment. *When developing a concrete instance of a specification theory, it is, therefore, beneficial to support both the behavioral and signal-flow approaches. In this respect, a first gap in the basic A/G contract formulations, stems from the fact that, in spite of their generality, they fall short of supporting signal-flow models that make a distinction between uncontrolled inputs and controlled outputs.*

Furthermore, an additional set of challenges arises when contracts are to be formulated and manipulated along the design flow, and across different abstraction levels. As mentioned before, there is no universal modeling formalisms that can capture every aspect of a complex cyber-physical system, and guarantee, at the same time, tractable analysis techniques. As a result, designers need to "decompose" the underlying system into semantically different models, by adopting the most convenient formalisms to represent different system portions or viewpoints at different abstraction levels, and the most suitable tools to analyze and synthesize them separately. However, the abstraction methods in compositional reasoning and contract-based design are prevalently defined in the context of a single formalism, e.g. using language inclusion, simulation relations, or compositional methods based on deduction. *In system design, abstractions should instead be able to bridge heterogeneous formalisms ("semantic heterogeneity") and heterogeneous decomposition architectures ("structural heterogeneity"), to make system analysis and synthesis tractable, by consistently combining different verification and synthesis results. In this respect, while A/G contracts promise to encompass any kind of formalism and decomposition, they fall short of supporting the correct transitions between them.*

This chapter aims to fill the above gaps, with the ultimate goal of developing a truly general contract framework for cyber-physical system (CPS) design that provides formal support to all the steps of our methodology. In the following, we summarize our approach to address both of the issues.

3.1.1 Contracts and Interfaces for Requirement Engineering

To shed light on the subtleties of contract compatibility and consistency for signal-flow models, our approach is to relate the theory of A/G contracts with the one of interfaces, which make a clear distinction between uncontrolled inputs and controlled outputs. To be concrete, we start from the theory of *synchronous relational interfaces* [197]. We choose stateless relational interfaces rather than other, more general interface theories, such as interface automata, as the former are simpler and can offer more intuitive support to our investigation. To correctly relate the two frameworks, we need to refer to a common "semantic domain." We therefore provide an operator which transforms a relational interface into an A/G contract, in the natural way. In particular, a relational interface represented as a formula ϕ on inputs and outputs is mapped into a set of behaviors representing the safety property that ϕ holds at every (synchronous) step. This can be concretely represented by the linear temporal logic (LTL) formula $\Box\phi$.

We then highlight differences and correspondences between key operators and relations in the two theories by studying their preservation properties under the above transformation. We show that, perhaps surprisingly, the basic operation of *serial composition* of interfaces is not preserved. Specifically, composing two interfaces $\mathcal{I}_1$ and $\mathcal{I}_2$, and then transforming the result to an A/G contract, is not equivalent to first transforming each of $\mathcal{I}_1$ and $\mathcal{I}_2$ to an A/G contract, and then composing the contracts. The reason for this is that the interface compatibility check is "built into" the interface composition operator, so that if the interfaces

are incompatible, the result of the composition is `F`. On the other hand, A/G contracts have no way of checking compatibility *a priori* during composition. Although compatibility can be checked *a posteriori* on the composite contract using the notion of *c-receptiveness* [33], the latter provides a yes/no answer and does not infer new environment assumptions, as in the case of interface composition.

To remedy this, we introduce an *assumption-projection* operator for A/G contracts. The latter eliminates ("hides") a given set of variables (only) from the assumption, using universal (i.e., game-theoretic) rather than the usual existential quantification. We show that with this hiding operator the transformation preserves the semantics of interface composition and compatibility. Moreover, the result can be extended to the notion of contract consistency, which is not explicitly formalized by relational interfaces[1], by defining a similar projection operator for the contract guarantees, thus incorporating the concept of *u-receptiveness* [33] into contract consistency. Unfortunately, LTL formulas are not generally closed under variable elimination (projection). It is, therefore, necessary to resort to a strictly more expressive extension of LTL, such as Quantified Linear Temporal Logic (QLTL) [190, 162], to implement this hiding operator. The satisfiability problem for QLTL has been shown to be decidable, but with *non-elementary* complexity [190].

We also show that our transformation preserves *refinement*, that is, interface refinement between interfaces $\mathcal{I}_1$ and $\mathcal{I}_2$ is equivalent to A/G contract refinement between the corresponding A/G contracts. However, another interesting operator, that of *conjunction* (also called *shared refinement* [197]) is not preserved. The reason is another crucial difference between the two frameworks. While A/G contracts reason about global behaviors of components, possibly spanning infinite sequences of reactions, relational interfaces can also capture punctual relations between the inputs and outputs of a component, at the granularity of a single reaction index. Therefore, computation of conjunction as the greatest lower bound (GLB) with respect to the refinement order, generates a smaller set of allowed environments and a larger set of guaranteed behaviors for A/G contracts, which translates into a tighter, less conservative, bound. As a result, the contract associated with the conjunction of interfaces $\mathcal{I}_1$ and $\mathcal{I}_2$ refines, but is generally different than, the conjunction of the contracts associated with $\mathcal{I}_1$ and $\mathcal{I}_2$.

Despite the proliferation of work on compositional theories in general, and interface and contract theories in particular, there is little work that attempts at drawing links between the existing frameworks, as we propose in this chapter. Benveniste et al. [34] propose a general "meta-theory" of contracts, expressed in terms of sets of implementations and environments, and from which both interface theories and A/G contracts can be instantiated. Following a similar approach, Bauer et al. [29] attempt at providing an abstract formalization of the notion of contracts by relating "specification theories" to "contract theories." Our approach is novel in that, instead of recurring to a common, more abstract, meta-theory, we aim to

[1]Since "assumptions" and "guarantees" are merged together in interface theories, the concept of interface compatibility is equivalent to both compatibility and consistency in contracts. In other words, an interface is compatible if and only if the associated contract (via the proposed transformation) is both compatible and consistent.

directly map interfaces to A/G contracts and, as a result, reveal some of the subtle differences in the two frameworks.

3.1.2 Contracts for Heterogeneous Refinement and Mapping

To truly support multi-level and multi-view design in a compositional way, we need to equip the general A/G contracts framework with relations and operations for abstraction/refinement between pairs of heterogeneous formalisms as well as pairs of heterogeneous modeling architectures. In this chapter, we introduce the concept of *heterogeneous refinement* to deal with the first form of heterogeneity, denoted as *semantic heterogeneity*; we further introduce the concept of *vertical contracts*, to encompass different notions of refinement and deal with the latter form of heterogeneity, denoted as as *structural heterogeneity*, at the same time as semantic heterogeneity.

In model-based verification, heterogeneous abstractions have been used in the past for specific pairs of formalisms, such as hybrid abstractions of nonlinear systems [96, 68], linear hybrid automata abstractions of linear hybrid systems [81], discrete abstractions of hybrid systems [12, 13, 57]. We presented some of the tools based on these abstractions in Chapter 2. By casting heterogeneous refinement within a generic A/G contract framework, we layout the foundations for a general compositional framework for heterogeneous abstraction that applies to any pair of heterogeneous formalisms.

Heterogeneous reactive systems can be compared and composed using the tagged-signal semantics [121]. This approach uses system trajectories or behaviors as a mathematical framework for creating relations between the semantics of different modeling formalisms. A formal framework that addresses the semantic heterogeneity of CPS by relating the semantics of different models defined in different formalisms using behaviors and their mappings has also been developed recently by Rajhans et al. [169]. In a similar spirit, we use mathematical functions between behavior domains as the semantic mappings between heterogeneous behaviors and contracts. However, in addition to verification, our intent is to provide support for other forms of heterogeneous refinement, such as synthesis and optimized mapping, as well as incorporate the concept of *orthogonalization of concerns*, such as communication-computation, function-architecture and behavior-performance [110].

At some point in the design flow, specifications must be realized by using resources. Resources can be pre-defined or newly developed components, including, e.g., computing units or communication media (networks, buses, and protocols). When deploying an application over a computing platform, in addition to the functional viewpoint, non-functional viewpoints (e.g., safety, timing, energy) are of importance as well. While it is still convenient to keep the separation between the "specification platform" used for initial prototyping and the supporting "execution platform" for deployment, the actual satisfaction of design requirements will heavily depend on the execution platform. It is often the case that the two platforms have distinct structural decompositions. Moreover, combining different viewpoints of the lower-level platform (e.g. timing and functional) may be necessary to effectively prove the correctness of the refinement of a single viewpoint (e.g. functional) of the higher-level

platform. In other words, refinement also depends on the "mapping" of the application into the execution platform.

In this chapter, we formalize this notion of mapping using *vertical contracts*. We first introduced and demonstrated the concept of vertical contracts in the context of platform-based design for analog and mixed-signal systems [152, 155]. Their generalization for system design, with application to control systems, was then advocated by Sangiovanni-Vincetelli et al. [177]. A formalization of vertical contracts in terms of contract conjunction was proposed by Benveniste et al. [34]. In this chapter, we build on this formalization to provide a richer vista on vertical contracts, showing that they can be expressed by using both composition and conjunction operators, based on the specific shape of the contracts attached to the different abstraction levels and viewpoints in the design. Such a generalization is key to enable any logical breakdown of complex system verification and synthesis tasks into arbitrary conjunctive and disjunctive combinations of smaller sub-tasks.

Finally, we observe that ontologies have been used in the past as a knowledge-management approach to combine verification or analysis results across heterogeneous models in a consistent way. For instance, lattice-based ontologies can be used to infer semantic relationships between elements of heterogeneous models [125]. However, contracts are a strictly more expressive framework than ontologies. Moreover, rather than treating verification activities as knowledge to be combined, we use logical combinations of verification and synthesis tasks, mediated by contracts, to develop complex hierarchies. In a similar spirit, the temporal logic of actions proof system deploys a proof manager that breaks down a complex verification task logically into proof obligations that are proved using theorem provers and satisfiability-modulo-theories solvers [55], but this framework is primarily aimed towards software systems, whereas our framework aims to rather develop a "design manager," supporting more general (e.g., continuous, hybrid) dynamics, non-deductive analysis techniques as well as synthesis and optimization methods.

3.1.3 Chapter Organization

The rest of the chapter is organized as follows. We discuss our mapping between contracts and relational interfaces in Section 3.2, and show its implications on contract compatibility and consistency in Section 3.3. Section 3.4 deals with heterogeneous refinement and vertical contracts, while Section 3.5 draws some conclusions.

3.2 Mapping Relational Interfaces into A/G Contracts

In this section, we first recall the salient parts of synchronous relational interfaces. Then, we detail our mapping between synchronous relational interfaces and A/G contracts. Relational interfaces have been proposed as an interface theory for synchronous systems that can capture functional relations between the inputs and the outputs of a component [197]. Input/output

relations are expressed as first-order logic formulas over the input and output variables. The developed theory supports two types of composition, serial connection and feedback, as well as refinement, compatibility and conjunction (denoted as shared refinement). On the other hand, as mentioned earlier, the A/G contract framework is more abstract in the sense that it does not predefine the type of behaviors and supports a richer composition algebra. Therefore, to establish our results, we concretely instantiate the A/G contract theory by considering a specific type of behaviors. Specifically, given a finite set of variables V, we define a behavior over V as an infinite sequence of valuations over V, $\rho = v_0 v_1 v_2 \cdots$. Then, in this section, we consider A/G contracts defined as triples (V, A, G) where A and G are sets of behaviors over V.

3.2.1 Background on Synchronous Relational Interfaces

For simplicity, we restrict ourselves to *stateless* interfaces. A (relational) interface is a tuple $\mathcal{I} = (X, Y, \phi)$ where X and Y are finite sets of input and output variables, respectively, and ϕ is a first-order logic formula on the variables in $X \cup Y$ [197]. The sets of input and output variables must be disjoint: $X \cap Y = \emptyset$. We assume that each variable in $V = X \cup Y$ ranges over the set of values $\mathcal{V}$. A *valuation* over V is a function $v : V \to \mathcal{V}$. A valuation v over V satisfies a formula ϕ over the same set of variables V, written $v \models \phi$, if replacing free variables in ϕ by their value as specified by v yields a formula that evaluates to `T`. A formula ϕ defines the following set of behaviors:

$$[\![\phi]\!] := \{v_0 v_1 v_2 \cdots \mid \forall i : v_i \models \phi\}.$$

Note that $[\![\phi]\!]$ is a safety property. By defining the above set of behaviors, ϕ governs the operation of the component specified by $\mathcal{I}$, evolving in a sequence of synchronous steps. At each step, a valuation over V must be found which satisfies ϕ.

Given interface $\mathcal{I} = (X, Y, \phi)$, the *input assumption* defined by ϕ is the formula $in(\phi) := \exists Y : \phi$, where $\exists Y : \phi$ is $\exists y_1 : \exists y_2 : \cdots \exists y_n : \phi$ when $Y = \{y_1, y_2, ..., y_n\}$. $in(\phi)$ characterizes the legal inputs. An input is considered illegal if there is no output which can satisfy ϕ for that input. Note that $in(\phi)$ is a formula on X only, as variables in Y have been eliminated by existential quantification. For example, if $X = \{x\}$, $Y = \{y\}$, and ϕ is $x \geq 0 \wedge y = x$, then $in(\phi)$ is $x \geq 0$. If ϕ is $x \geq 0 \to y = x$, then $in(\phi)$ is `T`.

3.2.1.1 Composition

Serial composition of two interfaces $\mathcal{I}_1 = (X_1, Y_1, \phi_1)$ and $\mathcal{I}_2 = (X_2, Y_2, \phi_2)$ can be defined provided that $\mathcal{I}_1$ and $\mathcal{I}_2$ are *disjoint*, i.e. all sets X_1, Y_1, X_2, Y_2 are pairwise disjoint, except possibly the pair Y_1, X_2. Let $V_c = Y_1 \cap X_2$. The interpretation is that variables in V_c are outputs of $\mathcal{I}_1$ which are connected to inputs of $\mathcal{I}_2$. Note that we allow V_c to be empty, in which case serial composition reduces to parallel composition (where no connections between the two interfaces exist). Then, the composite interface $\mathcal{I}_1 \rightsquigarrow \mathcal{I}_2$ is defined to be the interface

$$\mathcal{I}_1 \rightsquigarrow \mathcal{I}_2 := (X_1 \cup X_2 \setminus Y_1, Y_1 \cup Y_2, \phi)$$

where

$$\phi = \phi_1 \wedge \phi_2 \wedge \forall Y_1 : \big(\phi_1 \rightarrow in(\phi_2)\big).$$

$\mathcal{I}_1$ and $\mathcal{I}_2$ are said to be *compatible interfaces* if ϕ is satisfiable, i.e., if ϕ is not equivalent to `F`. Interface compatibility can then be checked while computing the serial composition.

3.2.1.2 Refinement

Given two interfaces $\mathcal{I}_1 = (X_1, Y_1, \phi_1)$ and $\mathcal{I}_2 = (X_2, Y_2, \phi_2)$, we say that $\mathcal{I}_1$ *refines* $\mathcal{I}_2$, written $\mathcal{I}_1 \sqsubseteq \mathcal{I}_2$, iff $X_1 \subseteq X_2$, $Y_1 \supseteq Y_2$ and the following formula is valid (i.e., true under all valuations):

$$in(\phi_2) \rightarrow \big(in(\phi_1) \wedge (\phi_1 \rightarrow \phi_2)\big).$$

3.2.1.3 Shared refinement

Two interfaces $\mathcal{I}_1 = (X, Y, \phi_1)$ and $\mathcal{I}_2 = (X, Y, \phi_2)$ are said to be *shared-refinable* if the following formula is true:

$$\forall X : \Big(\big(in(\phi_1) \wedge in(\phi_2)\big) \rightarrow \big(\exists Y : (\phi_1 \wedge \phi_2)\big)\Big).$$

This amounts to state that for every input that is legal in both $\mathcal{I}_1$ and $\mathcal{I}_2$, the corresponding sets of outputs of $\mathcal{I}_1$ and $\mathcal{I}_2$ must have a non-empty intersection. If $\mathcal{I}_1$ and $\mathcal{I}_2$ are shared-refinable, their shared refinement, denoted $\mathcal{I}_1 \sqcap \mathcal{I}_2$, is defined to be the interface $\mathcal{I}_1 \sqcap \mathcal{I}_2 := (X, Y, \phi_\sqcap)$, where

$$\phi_\sqcap := \big(in(\phi_1) \vee in(\phi_2)\big) \wedge \big(in(\phi_1) \rightarrow \phi_1\big) \wedge \big(in(\phi_2) \rightarrow \phi_2\big).$$

It can be shown that $\mathcal{I}_1 \sqcap \mathcal{I}_2$, when it exists, acts as the greatest lower bound (GLB) for the refinement relation, i.e. (i) $\mathcal{I}_1 \sqcap \mathcal{I}_2$ is guaranteed to refine both $\mathcal{I}_1$ and $\mathcal{I}_2$, and (ii) for any interface $\mathcal{I}'$ such that $\mathcal{I}' \sqsubseteq \mathcal{I}_1$ and $\mathcal{I}' \sqsubseteq \mathcal{I}_2$, we have $\mathcal{I}' \sqsubseteq (\mathcal{I}_1 \sqcap \mathcal{I}_2)$. The existence of such a GLB is an important feature for hierarchical component-based design supporting multiple viewpoints [33, 78, 95]. Shared refinement in relational interfaces has the same properties of conjunction of A/G contracts. However, shared refinement of interfaces is not always defined, whereas conjunction of A/G contracts is always defined as the GLB for the refinement preorder, which is guaranteed to exist.

3.2.2 Contract Associated with an Interface

We map interfaces into contracts based on the following definition.

Definition 3.2.1 (Contract Associated with an Interface)**.** *An interface* $\mathcal{I} = (X, Y, \phi)$ *can be transformed into a contract* $\mathcal{C} = \mathcal{F}(\mathcal{I}) = (V, A, G)$ *where*

$$V := X \cup Y, \quad A := \Box in(\phi), \quad G := \Box in(\phi) \rightarrow \Box\phi.^2$$

We call $\mathcal{C}$ *the contract* associated *with* $\mathcal{I}$ *under the transformation* $\mathcal{F}$*.*

Even though $in(\phi)$ is a formula over only the set of input variables X, when we define A we choose to interpret $in(\phi)$ over the entire set of variables $V = X \cup Y$. In fact, both A and G in a contract are defined as behaviors over the same set of variables. Moreover, we conveniently express the sets of behaviors in A and G as LTL formulas, where $\Box\phi$ denotes the set of behaviors $[\![\phi]\!]$. A relational interface represented as a formula ϕ on inputs and outputs is then mapped into sets of behaviors representing the safety property that ϕ holds at every (synchronous) step, under the assumption that $in(\phi)$ holds at every step.

The contract $\mathcal{F}(\mathcal{I})$ in Definition 3.2.1 preserves the semantics of the associated interface $\mathcal{I}$. Any legal environment of $\mathcal{F}(\mathcal{I})$ provides, at every step, a legal input for $\mathcal{I}$ (while accepting any output from $\mathcal{I}$), and *vice versa*. On the other hand, any system satisfying ϕ at every step is an implementation for $\mathcal{F}(\mathcal{I})$, since it satisfies $\Box\phi$ in any context in which $\Box in(\phi)$ holds. Finally, by definition, contract $\mathcal{F}(\mathcal{I})$ is in saturated form. In what follows, we analyze the properties of serial composition, refinement and conjunction in both interfaces and contracts with respect to the proposed transformation.

3.2.3 Serial Composition and Compatibility

In this section, by abuse of notation, we use the parallel composition operator $\otimes$, defined in Section 2.3.2, to also denote the serial composition of contracts. As shown in Section 2.3.2, serial composition can be obtained from the parallel composition of contracts, in which every pair of shared variables are given the same name to denote the presence of an interconnection. However, while parallel composition is commutative, serial composition is generally non-commutative.

To establish a correspondence between interfaces and contracts, we would like serial composition and compatibility to be preserved under $\mathcal{F}$, i.e., for the interfaces $\mathcal{I}_1$ and $\mathcal{I}_2$, $\mathcal{F}(\mathcal{I}_1 \rightsquigarrow \mathcal{I}_2) = \mathcal{F}(\mathcal{I}_1) \otimes \mathcal{F}(\mathcal{I}_2)$ to hold. However, this is not true in general, as shown by the following example.

Example 12. *Consider the interfaces* $\mathcal{I}_1 = (\{x\}, \{y\}, \mathtt{T})$ *and* $\mathcal{I}_2 = (\{y\}, \emptyset, y \geq 0)$*, shown in Figure 3.1(a), where* $x, y \in \mathbb{R}$*. We have* $\mathcal{F}(\mathcal{I}_1) = (\{x, y\}, \mathtt{T}, \mathtt{T})$ *and* $\mathcal{F}(\mathcal{I}_2) = (\{x, y\}, \Box(y \geq 0), \mathtt{T})$*. Moreover, since* $\mathcal{I}_1 \rightsquigarrow \mathcal{I}_2 = (\{x\}, \{y\}, \mathtt{F})$*, we have* $\mathcal{F}(\mathcal{I}_1 \rightsquigarrow \mathcal{I}_2) = (\{x, y\}, \mathtt{F}, \mathtt{T})$*. On the other hand, we also obtain* $\mathcal{F}(\mathcal{I}_1) \otimes \mathcal{F}(\mathcal{I}_2) = (\{x, y\}, \Box(y \geq 0), \mathtt{T})$*, which is clearly not equal to* $\mathcal{F}(\mathcal{I}_1 \rightsquigarrow \mathcal{I}_2)$*.*

[2] $\Box$ takes precedence over $\rightarrow$, so $\Box in(\phi) \rightarrow \Box\phi$ means $\big(\Box in(\phi)\big) \rightarrow \Box\phi$. Moreover, in this and the following subsections, we use A and G to interchangeably denote sets of behaviors as well as the logic formulas specifying these behaviors.

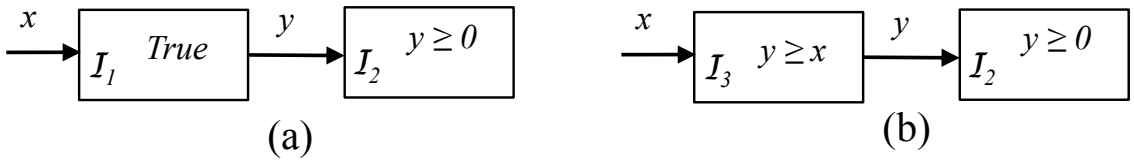


Figure 3.1: Pictorial representation of the relational interfaces in Example 12 (a) and Example 13 (b).

The difference highlighted by Example 12 can be intuitively explained by the incompatibility of $\mathcal{I}_1$ and $\mathcal{I}_2$. This is correctly expressed by $\phi_{\mathcal{I}_1 \rightsquigarrow \mathcal{I}_2}$ being $\mathtt{F}$ and reflected into the assumptions of $\mathcal{F}(\mathcal{I}_1 \rightsquigarrow \mathcal{I}_2)$, which are also $\mathtt{F}$. The contract $\mathcal{F}(\mathcal{I}_1 \rightsquigarrow \mathcal{I}_2)$ is also incompatible, i.e. any component satisfying $\mathcal{F}(\mathcal{I}_1 \rightsquigarrow \mathcal{I}_2)$ cannot be hosted by any environment. However, such incompatibility is not immediately detected using $\mathcal{F}(\mathcal{I}_1) \otimes \mathcal{F}(\mathcal{I}_2)$, which seems to indicate that any sequence y_n satisfying $y_n \geq 0$ for all $n \in \mathbb{N}$ is admitted. Only after observing that y is a controlled variable, we can finally conclude that $\mathcal{F}(\mathcal{I}_1) \otimes \mathcal{F}(\mathcal{I}_2)$ is incompatible, since its assumptions are not y-receptive.

As a second attempt, we may try to prove that serial composition is preserved provided the interfaces are compatible. Example 13 shows that this is not the case either.

Example 13. *Consider the interfaces* $\mathcal{I}_3 = (\{x\}, \{y\}, y \geq x)$ *and* $\mathcal{I}_2 = (\{y\}, \emptyset, y \geq 0)$*, shown in Figure 3.1(b). We have* $\mathcal{F}(\mathcal{I}_3) = (\{x, y\}, \mathtt{T}, \Box(y \geq x))$*,* $\mathcal{F}(\mathcal{I}_2) = (\{x, y\}, \Box(y \geq 0), \mathtt{T})$*,* $\mathcal{I}_3 \rightsquigarrow \mathcal{I}_2 = (\{x\}, \{y\}, x \geq 0 \wedge y \geq x)$*, and*

$$\mathcal{F}(\mathcal{I}_3 \rightsquigarrow \mathcal{I}_2) = (\{x, y\}, \Box(x \geq 0), \Box(x \geq 0) \rightarrow \Box(y \geq x)).$$

On the other hand, we also obtain

$$\mathcal{F}(\mathcal{I}_3) \otimes \mathcal{F}(\mathcal{I}_2) = (\{x, y\}, \Box(y \geq x) \rightarrow \Box(y \geq 0), \Box(y \geq x)),$$

which is clearly not equal to $\mathcal{F}(\mathcal{I}_3 \rightsquigarrow \mathcal{I}_2)$*. In fact, the sequence* (x_n, y_n) *where* $x_n = -1$ *and* $y_n = -3$ *for all* $n \in \mathbb{N}$ *satisfies the assumptions of* $\mathcal{F}(\mathcal{I}_3) \otimes \mathcal{F}(\mathcal{I}_2)$ *but does not satisfy the ones of* $\mathcal{F}(\mathcal{I}_3 \rightsquigarrow \mathcal{I}_2)$*.*

Again, we see that the assumptions of $\mathcal{F}(\mathcal{I}_3) \otimes \mathcal{F}(\mathcal{I}_2)$ in Example 13 refer to output variables, and do not contain the important new assumption $x \geq 0$ induced by interface composition, and which is crucial to guarantee interface compatibility. Note that we can still conclude that $\mathcal{F}(\mathcal{I}_3) \otimes \mathcal{F}(\mathcal{I}_2)$ is indeed compatible, since its assumptions are y-receptive. However, we are also interested in inferring the set of environments that is allowed by the composite contract, as captured by the new assumption $\Box x \geq 0$. To obtain this, we introduce a new projection operation on contracts, which we call *assumption projection* (AP).

3.2.4 Assumption Projection

Definition 3.2.2 (Assumption Projection)**.** *Given a contract $\mathcal{C} = (V, A, G)$, and a subset $W \subseteq V$, the* assumption projection *of $\mathcal{C}$ with respect to W (AP_W) returns the new saturated contract*

$$\mathrm{AP}_W(\mathcal{C}) = (V, \forall W : A, (\forall W : A) \rightarrow G).$$

We use the fact that the universal quantifier is commutative and associative to lift it to sets of variables in Definition 3.2.2, so that $\forall W : A := (\forall w_1 : \forall w_2 : \ldots : \forall w_n : A)$ when $W = \{w_1, w_2, \ldots, w_n\}$. Moreover, when the assumptions are expressed by an LTL formula, universal quantification is meant over sequences of valuations over the variables in W [162]. We are now ready to state the following theorem, which relates serial composition of interfaces with serial composition of contracts.

Theorem 3.2.3 (Assumption Projection Mapping)**.** *Given two disjoint relational interfaces $\mathcal{I}_1$ and $\mathcal{I}_2$, with sets of output variables Y_1 and Y_2, respectively, we have*

$$\mathcal{F}(\mathcal{I}_1 \rightsquigarrow \mathcal{I}_2) = \mathrm{AP}_{Y_1 \cup Y_2}(\mathcal{F}(\mathcal{I}_1) \otimes \mathcal{F}(\mathcal{I}_2)). \tag{3.1}$$

Moreover, $\mathcal{I}_1$ and $\mathcal{I}_2$ are compatible iff $\mathrm{AP}_{Y_1 \cup Y_2}(\mathcal{F}(\mathcal{I}_1) \otimes \mathcal{F}(\mathcal{I}_2))$ is compatible.

According to Theorem 3.2.3, the contract associated with the composition of two interfaces is equivalent to the assumption projection contract of the composition of the associated contracts with respect to the output variables, i.e. we can still map the contract associated with the composition of two interfaces $\mathcal{F}(\mathcal{I}_1 \rightsquigarrow \mathcal{I}_2)$ to the composition of the associated contracts $\mathcal{F}(\mathcal{I}_1) \otimes \mathcal{F}(\mathcal{I}_2)$ only after applying the AP operator. Before proving Theorem 3.2.3, we introduce the following lemma, which will be used in the proof.

Lemma 3.2.4. *Given the interfaces $\mathcal{I}_1 = (X_1, Y_1, \phi_1)$ and $\mathcal{I}_2 = (X_2, Y_2, \phi_2)$, let $\psi = \square(\forall Y_1 : \phi_1 \rightarrow in(\phi_2))$, and $\psi' = (\forall Y_1 : \square\phi_1 \rightarrow \square in(\phi_2))$. Then, if $\square(in(\phi_1))$ is* `T`*, we have $\psi \leftrightarrow \psi'$.*

Proof (Lemma 3.2.4). Suppose first that ψ is `T`, and suppose that on all sequences $y_{1,n}$ of valuations over Y_1, $\square\phi_1$ holds. Then, for all n, for all valuations $(x_{1,n}, x_{2,n}, y_{1,n})$ over (X_1, X_2, Y_1), we have $(x_{1,n}, x_{2,n}, y_{1,n}) \models \phi_1$. Hence, by ψ, we also have that for all n, for all the valuations over (X_1, X_2, Y_1), $(x_{1,n}, x_{2,n}, y_{1,n}) \models in(\phi_2)$. This implies that $\square in(\phi_2)$ is also valid for all sequences of valuations over Y_1, and ψ' is `T`. Therefore, we conclude that $\psi \rightarrow \psi'$.

To prove that $\psi' \rightarrow \psi$, we now assume that ψ is `F`, and prove that ψ' must also be `F`. In fact, if ψ is `F`, then there exists a sequence $(x_{1,k}, x_{2,k})$ of valuations over (X_1, X_2), an index $i \in \mathbb{N}$ and a valuation y^* over Y_1 such that $(x_{1,i}, x_{2,i}, y^*) \models \phi_1$ and $(x_{1,i}, x_{2,i}, y^*) \not\models in(\phi_2)$. Consider such a sequence $(x_{1,k}, x_{2,k})$. Then, since $\square in(\phi_1)$ holds by hypothesis, we know that, for all k, it is possible to find $\hat{y}_{1,k}$ such that $(x_{1,k}, \hat{y}_{1,k}) \models \phi_1$. Therefore, starting from $(x_{1,k}, x_{2,k})$, we can construct a new sequence $s_k = (x_{1,k}, x_{2,k}, y_{1,k})$ such that $\forall k \neq i$, $y_{1,k} = \hat{y}_{1,k}$, and for $k = i$, $y_{1,i} = y^*$. By construction, $s_k \models \square\phi_1$ but $s_k \not\models \square in(\phi_2)$, i.e. s_k falsifies ψ'. We can therefore conclude $\neg\psi \rightarrow \neg\psi'$, which is what we wanted to prove. $\square$

We can now prove Theorem 3.2.3.

Proof (Theorem 3.2.3). Both the left and right-hand side contracts $\mathcal{C}_L$ and $\mathcal{C}_R$ in (3.1) are in saturated form by definition of $\mathcal{F}$ and of AP. To prove that $\mathcal{C}_L$ and $\mathcal{C}_R$ are equal we need to prove that they have the same assumption and guarantee sets. We first compute assumptions and guarantees for $\mathcal{C}_R$. By applying (2.2) and (2.3) and the definition of $\mathcal{F}$ we obtain:

$$G_\otimes = (\Box in(\phi_1) \rightarrow \Box\phi_1) \wedge (\Box in(\phi_2) \rightarrow \Box\phi_2) \tag{3.2}$$

$$\begin{aligned} A_\otimes &= (\Box in(\phi_1) \wedge \Box in(\phi_2)) \vee \neg G_\otimes \\ &= \Box(in(\phi_1) \wedge in(\phi_2)) \vee (\Box in(\phi_1) \wedge \neg\Box\phi_1) \vee (\Box in(\phi_2) \wedge \neg\Box\phi_2) \end{aligned} \tag{3.3}$$

where $A_\otimes$ and $G_\otimes$ are the assumptions and guarantees of $\mathcal{F}(\mathcal{I}_1) \otimes \mathcal{F}(\mathcal{I}_2)$. Finally, after assumption projection, we obtain:

$$\begin{aligned} A_R &= \forall Y_1 \forall Y_2 : A_\otimes \\ &= \forall Y_1 : \Box(in(\phi_1) \wedge in(\phi_2)) \vee (\Box in(\phi_1) \wedge \neg\Box\phi_1) \vee (\forall Y_2 : (\Box in(\phi_2) \wedge \neg\Box\phi_2)) \\ &= \forall Y_1 : \Box(in(\phi_1) \wedge in(\phi_2)) \vee (\Box in(\phi_1) \wedge \neg\Box\phi_1) \\ &= \forall Y_1 : \Box in(\phi_1) \wedge (\Box in(\phi_2) \vee \neg\Box\phi_1) \\ &= \Box in(\phi_1) \wedge (\forall Y_1 : \Box\phi_1 \rightarrow \Box in(\phi_2)) \end{aligned} \tag{3.4}$$

$$\begin{aligned} G_R &= A_R \rightarrow G_\otimes \\ &= \Box in(\phi_1) \wedge (\forall Y_1 : \Box\phi_1 \rightarrow \Box in(\phi_2)) \rightarrow (\Box\phi_1 \vee \neg\Box in(\phi_1)) \wedge (\Box\phi_2 \vee \neg\Box in(\phi_2)) \end{aligned} \tag{3.5}$$

Consider now the assumptions of $\mathcal{C}_L$. We obtain:

$$\begin{aligned} A_L = \Box in(\phi) &= \Box\left[\exists Y_1 \exists Y_2 : \phi_1 \wedge \phi_2 \wedge (\forall Y_1 : \phi_1 \rightarrow in(\phi_2))\right] \\ &= \Box\left[(\forall Y_1 : \phi_1 \rightarrow in(\phi_2)) \wedge (\exists Y_1 : \phi_1 \wedge in(\phi_2))\right] \\ &= \Box\left[(\forall Y_1 : \phi_1 \rightarrow in(\phi_2)) \wedge in(\phi_1)\right] \\ &= \Box(\forall Y_1 : \phi_1 \rightarrow in(\phi_2)) \wedge \Box in(\phi_1) \end{aligned} \tag{3.6}$$

while for G_L we obtain

$$G_L = \Box(\forall Y_1 : \phi_1 \rightarrow in(\phi_2)) \wedge \Box in(\phi_1) \rightarrow \Box(\phi_1 \wedge \phi_2 \wedge (\forall Y_1 : \phi_1 \rightarrow in(\phi_2))). \tag{3.7}$$

The equivalence of the assumptions A_L and A_R directly descends from Lemma 3.2.4. To prove the equivalence of G_L and G_R it is enough to prove that, if A_L or A_R is $\mathtt{T}$, then

$$(\Box in(\phi_1) \rightarrow \Box\phi_1) \wedge (\Box in(\phi_2) \rightarrow \Box\phi_2) \leftrightarrow \Box(\phi_1 \wedge \phi_2). \tag{3.8}$$

Clearly, if the formula on the left side of the double implication in (3.8) is $\mathtt{T}$, the formula on the right side is also trivially $\mathtt{T}$ when A_R and A_L are $\mathtt{T}$. Suppose now that the left-hand

side of (3.7) is $\mathtt{T}$. Since A_L and A_R are $\mathtt{T}$ then $\Box in(\phi_1)$ is $\mathtt{T}$, which implies $\Box\phi_1$ is $\mathtt{T}$. On the other hand, by A_L and A_R being again $\mathtt{T}$, we also have

$$\Box(\forall Y_1 : \phi_1 \rightarrow in(\phi_2)) \wedge \Box\phi_1 \rightarrow \Box in(\phi_2).$$

This allows us to conclude that $\Box\phi_2$ is also $\mathtt{T}$ and finally (3.8) holds. We have therefore proved (3.1).

Let now $\phi = \phi_1 \wedge \phi_2 \wedge (\forall Y_1 : \phi_1 \rightarrow in(\phi_2))$ be the formula associated with $\mathcal{I}_1 \rightsquigarrow \mathcal{I}_2$. $\mathcal{I}_1$ and $\mathcal{I}_2$ are compatible if and only if ϕ is satisfiable. On the other hand, $\mathrm{AP}_{Y_1 \cup Y_2}(\mathcal{F}(\mathcal{I}_1) \otimes \mathcal{F}(\mathcal{I}_2))$ is compatible if and only if its assumptions A_R are satisfiable. Then, to prove the last statement of the theorem, we need to prove that ϕ is satisfiable if and only if A_R is satisfiable. This can be directly inferred from the fact that $A_R = A_L = \Box in(\phi)$. In fact, $\Box in(\phi)$ is satisfiable if and only if $in(\phi)$ is satisfiable, i.e. if and only if ϕ is satisfiable, which concludes our proof. □

The assumption projection mapping allows preserving the semantics of interface composition; we demonstrate its application to Examples 8 and 9 discussed in Chapter 2, and Examples 12 and 13 in Section 3.2.3. In Example 8, while computing the assumptions of $\mathrm{AP}_y(\mathcal{C}_{L1} \otimes \mathcal{C}_D)$, we obtain, as expected, $A_{ex1} = (\forall y : y \neq 0) = \mathtt{F}$, which corresponds to $\mathcal{C}_{L1}$ and $\mathcal{C}_D$ being incompatible. A similar result is obtained in Example 12, where $A_{ex3} = (\forall y : \Box(y \geq 0)) = \mathtt{F}$ are the assumptions of $\mathrm{AP}_y(\mathcal{F}(\mathcal{I}_1) \otimes \mathcal{F}(\mathcal{I}_2))$. By applying assumption projection to $\mathcal{C}_{L2} \otimes \mathcal{C}_D$ in Example 9, we instead obtain

$$\begin{aligned} A_{ex2} = \forall y : y \neq 0 \vee x \geq y &= \neg\ (\exists y :\ (y = 0 \wedge x < y)) \\ &= \neg\ ((\exists y :\ y = 0) \wedge x < 0) \\ &= x \geq 0, \end{aligned}$$

which corresponds to the desired set of legal environments for the composite specification. Finally, in Example 13, we need to compute $A_{ex4} = (\forall y : \Box(y \geq x) \rightarrow \Box(y \geq 0))$. To do this, it is convenient to exchange the order between the quantifier and the temporal construct, by using Lemma 3.2.4, with $\psi = \Box(\forall y : (y \geq x) \rightarrow (y \geq 0))$ and $\psi' = A_{ex4} = (\forall y : \Box(y \geq x) \rightarrow \Box(y \geq 0))$. Then, since the hypothesis of the lemma is satisfied, we conclude that ψ is equivalent to ψ' and, in particular,

$$\begin{aligned} A_{ex4} = \psi &= \Box(\forall y : (y \geq x) \rightarrow (y \geq 0)) \\ &= \Box\neg\ (\exists y :\ (y \geq x) \wedge (0 > y)) \\ &= \Box\neg\ (\exists y :\ x \leq y < 0) = \Box\neg(x < 0) = \Box x \geq 0, \end{aligned}$$

which, as expected, preserves the equivalence with the contract associated with the composite interface.

3.2.5 Implementing Assumption Projection in Temporal Logic

Assumption projection hides the controlled (output) variables of a composite contract from its assumptions, thus enabling preservation of serial composition and compatibility between interfaces and their associated contracts. However, we observe that this operator is not straightforward to implement, since LTL is not closed under projection [205]. For instance, consider the LTL formula ϕ over two Boolean variables s and p:

$$\phi := p \wedge \Box(s \rightarrow p) \wedge \Box(s \rightarrow \bigcirc\neg s) \wedge \Box(\neg s \rightarrow \bigcirc s).$$

It can be shown that there is no LTL formula over p that characterizes exactly the set of infinite traces obtained by projecting the traces characterized by ϕ onto the p variable.

It is, however, possible to resort to an extension of (propositional) LTL, namely Quantified (Propositional) Linear Temporal Logic (QLTL/QPTL), which introduces quantification over propositions [190, 162]. Having an expressive power equal to that of ω-automata, QLTL is strictly more expressive than LTL, and has been used for formulating and verifying refinement relations between reactive systems or programs [109]. However, we may pay for the augmented expressive power with a substantial increase in complexity. In fact, checking compatibility between two (Q)LTL A/G contracts after assumption projection can always be reduced to solving a satisfiability problem for QLTL, which is decidable with *non-elementary* complexity [190].

Every QLTL formula can be written in *normal form* as

$$(Q_1 p_1 Q_2 p_2 \ldots Q_k p_k : \varphi), \tag{3.9}$$

where $\{Q_1, Q_2, \ldots, Q_k\}$ is a finite sequence of existential or universal quantifiers, $P = \{p_1, p_2, \ldots, p_k\}$ a finite sequence of propositional variables, and φ is a quantifier-free formula. For each alternation of existential and universal quantifiers in (3.9) the space complexity increases by exactly one exponential. For instance, a QLTL formula of the form $(\exists P : \varphi)$, where φ is an LTL formula of size n, is said to be in the set Σ_1^{QLTL}. The satisfiability problem for this set of formulas, which are also denoted as Existentially Quantified LTL (EQLTL) formulas, is PSPACE-complete, like that for LTL. On the other hand, a formula of the form $(\forall P : \varphi)$ is instead said to be in Π_1^{QLTL}. For this set of formulas the satisfiability problem becomes complete for SPACE(2^n). A sound and complete proof system for QLTL is provided by French and Reynolds [83].

Finally, we may be tempted to assimilate the assumption-projection operator with the *elimination* (or *hiding*) operator introduced by Benveniste et al. [33, 35], in that they both allow hiding internal variables from the assumptions of a composite contract. For a contract $\mathcal{C} = (V, A, G)$ and a variable $p \in V$, the elimination of p in $\mathcal{C}$ is given by

$$[\mathcal{C}]_p = (V \setminus \{p\}, \forall p : A, \exists p : G),$$

where A and G are seen as predicates or logic formulas. However, unlike elimination, in assumption-projection, quantification is performed only on the contract assumptions, with-

out the intent of eliminating variables from the entire contract. Moreover, assumption-projection performs universal quantification over all the output variables, rather than just the internal ones.

3.2.6 Refinement

While $\mathcal{F}$ does not generally preserve serial composition, it preserves refinement, i.e. the mapping is monotonous, as the following theorem shows.

Theorem 3.2.5 (Refinement Preservation)**.** *Given two relational interfaces $\mathcal{I}_1$ and $\mathcal{I}_2$, then $\mathcal{I}_1 \sqsubseteq \mathcal{I}_2$ if and only if $\mathcal{F}(\mathcal{I}_1) \preceq \mathcal{F}(\mathcal{I}_2)$.*

Proof. Let $\mathcal{I}_1 = (X_1, Y_1, \phi_1)$ and $\mathcal{I}_2 = (X_2, Y_2, \phi_2)$. By definition of refinement, we recall that $\mathcal{I}_1 \sqsubseteq \mathcal{I}_2$ if and only if $(in(\phi_2) \rightarrow in(\phi_1) \wedge (\phi_1 \rightarrow \phi_2))$ is valid or, equivalently, the following two formulas

$$in(\phi_2) \rightarrow in(\phi_1) \tag{3.10}$$

$$in(\phi_2) \wedge \phi_1 \rightarrow \phi_2 \tag{3.11}$$

are both valid. Moreover, by definition of $\mathcal{F}$, we have

$$\mathcal{F}(\mathcal{I}_1) = (Y_1 \cup X_2, \Box in(\phi_1), \Box in(\phi_1) \rightarrow \Box\phi_1)$$

$$\mathcal{F}(\mathcal{I}_2) = (Y_1 \cup X_2, \Box in(\phi_2), \Box in(\phi_2) \rightarrow \Box\phi_2).$$

We first prove that $\mathcal{I}_1 \sqsubseteq \mathcal{I}_2 \rightarrow \mathcal{F}(\mathcal{I}_1) \preceq \mathcal{F}(\mathcal{I}_2)$. Let A_i and G_i be, respectively, the assumptions and the guarantees of $\mathcal{F}(\mathcal{I}_i)$. We need to show that formulas (3.10) and (3.11) imply $A_2 \rightarrow A_1$ and $G_1 \rightarrow G_2$. Assume $A_2 = \Box in(\phi_2)$ is `T`, then, by (3.10), $A_1 = \Box in(\phi_1)$ is also `T`; therefore, $A_2 \rightarrow A_1$. Assume now that G_1 is `T`, i.e. either $\Box in(\phi_1)$ is `F` or $\Box\phi_1$ is `T`. If $\Box in(\phi_1)$ is `F`, then from $A_2 \rightarrow A_1$, $\Box in(\phi_2)$ is also `F`, which makes G_2 `T`. If $\Box\phi_1$ is `T`, then, by (3.11), we conclude $\Box in(\phi_2) \rightarrow \Box\phi_2$, hence G_2 is again `T`. We therefore conclude that $G_1 \rightarrow G_2$.

We now prove that if $\mathcal{F}(\mathcal{I}_1) \preceq \mathcal{F}(\mathcal{I}_2)$, i.e. $A_2 \rightarrow A_1$ and $G_1 \rightarrow G_2$, then (3.10) and (3.11) are valid. To do so, we assume instead that $\mathcal{I}_1 \not\sqsubseteq \mathcal{I}_2$ and show that $\mathcal{F}(\mathcal{I}_1) \not\preceq \mathcal{F}(\mathcal{I}_2)$. In fact, if (3.10) is not valid, then we can create a sequence x_n of valuations over X_2 and an index i such that $x_n \models in(\phi_2)$ for all n, and $x_i \not\models in(\phi_1)$. Then, for such a sequence, $\Box in(\phi_2)$ is `T` while $\Box in(\phi_1)$ is `F`, which means that $A_2 \rightarrow A_1$ is not valid. Similarly, assume (3.11) is not valid; then we can create a sequence of valuations (x_n, y_n) for the variables in $X_2 \cup Y_1$ and an index i such that $(x_n, y_n) \models in(\phi_2)$ and $(x_n, y_n) \models \phi_1$ for all n, while $(x_i, y_i) \not\models \phi_2$. However, this implies that $\Box\phi_1$, hence G_1 is `T` while G_2 is `F`, since $\Box in(\phi_2)$ is `T` without $\Box\phi_2$ being `T`. Therefore, $G_1 \rightarrow G_2$ is also not valid, which allows us to conclude $(\mathcal{I}_1 \not\sqsubseteq \mathcal{I}_2) \rightarrow (\mathcal{F}(\mathcal{I}_1) \not\preceq \mathcal{F}(\mathcal{I}_2))$, as we wanted to prove. □

To enable compositional methods in system design, it is useful to investigate whether refinement is preserved by composition. For both relational interfaces and A/G contracts

refinement is preserved by parallel composition and serial composition [197, 33]. For instance, given the A/G contracts $\mathcal{C}_1, \mathcal{C}_2, \mathcal{C}'_1, \mathcal{C}'_2$, if $\mathcal{C}'_1 \preceq \mathcal{C}_1$, $\mathcal{C}'_2 \preceq \mathcal{C}_2$ and $\mathcal{C}_1$ is compatible with $\mathcal{C}_2$, we are allowed to conclude that $\mathcal{C}'_1$ is also compatible with $\mathcal{C}'_2$ and $\mathcal{C}'_1 \otimes \mathcal{C}'_2 \preceq \mathcal{C}_1 \otimes \mathcal{C}_2$. Therefore, compatible contracts can be independently refined, which is key to enable top-down incremental design, by iteratively decomposing a system-level contract $\mathcal{C}$ into sub-system contracts $\mathcal{C}_i$ for further independent development.

However, refinement is not always preserved by *feedback composition* in both frameworks. In relational interfaces, there is no composition operator that supports feedback loops for stateless interfaces. For feedback, an interface is required to be *Moore* with respect to the input variables involved in the connection. In a Moore interface, if an output y is fed-back to an input x, then y may only depend on state variables and on the current value of the input variables which are not connected to it, i.e. the ones in $X \setminus \{x\}$ [197]. This definition, inspired by Moore machines, allows forming feedback loops without creating causality cycles. As a result, in relational interfaces, feedback preserves refinement only if the interfaces are Moore with respect to the input variables involved in the connection.

Mapping feedback composition of relational interfaces into A/G contracts would require dealing with stateful interfaces, which is out of the scope of this thesis. In what follows, we discuss just one property of interest. Since feedback connection in A/G contracts can be defined based on the parallel composition operator, contract refinement is preserved by feedback composition. In Theorem 3.2.7, we show that this is also the case for the contracts associated with two relational interfaces, even if the original interfaces are not Moore. To do this, we first provide a definition of feedback which extends the one in Section 2.3.2 to LTL A/G contracts.

Definition 3.2.6 (Feedback Composition of LTL A/G Contracts)**.** *Given a contract $\mathcal{C} = (V, A, G)$ and a feedback connection $\kappa = (x, y) \in V^2$ on $\mathcal{C}$, let $\mathcal{C}_{id}$ be the contract defined as $\mathcal{C}_{id} = (\{x, y\}, \mathtt{T}, \square(x = y))$. Then, κ defines a new contract $\kappa(\mathcal{C}) := \mathcal{C} \otimes \mathcal{C}_{id}$[3].*

Theorem 3.2.7 (Refinement under Feedback Composition)**.** *Let $\mathcal{I}_1 = (X, Y, \phi_1)$ and $\mathcal{I}_2 = (X, Y, \phi_2)$ be two relational interfaces and $\kappa = (x, y) \in X \times Y$ a feedback connection on the associated contracts $\mathcal{F}(\mathcal{I}_1)$ and $\mathcal{F}(\mathcal{I}_2)$, then*

$$(\mathcal{I}_1 \sqsubseteq \mathcal{I}_2) \rightarrow (\kappa(\mathcal{F}(\mathcal{I}_1)) \preceq \kappa(\mathcal{F}(\mathcal{I}_2)))\,, \tag{3.12}$$

provided that $\kappa(\mathcal{F}(\mathcal{I}_2))$ is compatible.

Proof. By Theorem 3.2.5, we know that if $\mathcal{I}_1 \sqsubseteq \mathcal{I}_2$, then $\mathcal{F}(\mathcal{I}_1) \preceq \mathcal{F}(\mathcal{I}_2)$. By definition of κ, we also have $\kappa(\mathcal{F}(\mathcal{I}_1)) = \mathcal{F}(\mathcal{I}_1) \otimes \mathcal{C}_{id}$ and $\kappa(\mathcal{F}(\mathcal{I}_2)) = \mathcal{F}(\mathcal{I}_2) \otimes \mathcal{C}_{id}$, $\mathcal{C}_{id}$ being the contract $(\{x, y\}, \mathtt{T}, \square(x = y))$. Then, by the property of independent implementation of the parallel composition of contracts [34], if $\kappa(\mathcal{F}(\mathcal{I}_2))$ is compatible, we can conclude that $\kappa(\mathcal{F}(\mathcal{I}_1))$ is also compatible and $\kappa(\mathcal{F}(\mathcal{I}_1)) \preceq \kappa(\mathcal{F}(\mathcal{I}_2))$, as we wanted to show. □

[3] Although x is not renamed as y in this case, the definition is consistent with the one in Section 2.3.2; x remains in the variable set of $\mathcal{C}$, but is bound to be identical to y.

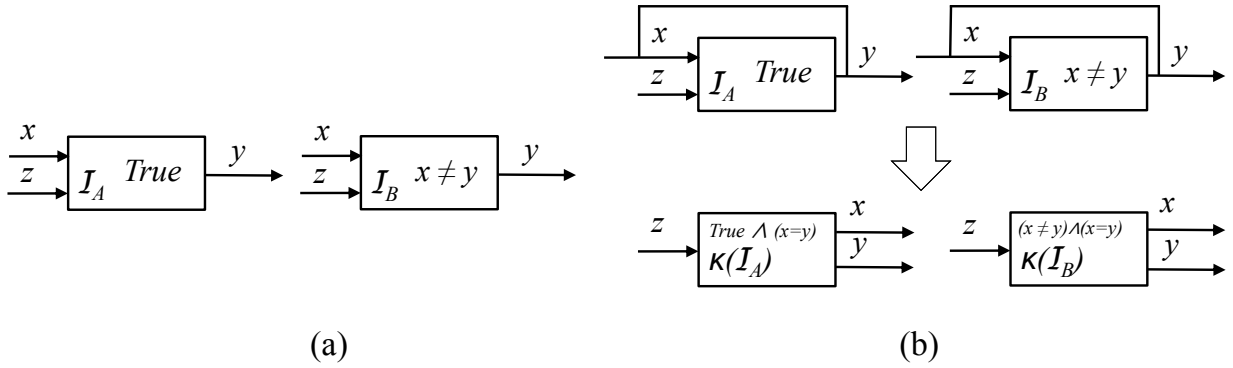


Figure 3.2: Configurations considered in Example 14.

As observed above, (3.12) holds even if $\mathcal{I}_1$ and $\mathcal{I}_2$ are not Moore with respect to x. A similar definition of feedback as the one stated in Definition 3.2.6 can also be extended to non-More interfaces as follows.

Definition 3.2.8 (Feedback for Non-Moore Interface)**.** *Given an interface $\mathcal{I} = (X, Y, \phi)$, a feedback connection $\kappa = (x, y) \in X \times Y$ defines a new interface $\kappa(\mathcal{I}) = (X \setminus \{x\}, Y \cup \{x\}, \phi \wedge (x = y))$.*

However, if $\mathcal{I}_1$ and $\mathcal{I}_2$ are not Moore, $\kappa(\mathcal{I}_1) \sqsubseteq \kappa(\mathcal{I}_2)$ is not guaranteed. This may happen either because $\phi_{\kappa(\mathcal{I}_1)}$ is $\mathtt{F}$ or because $\phi_{\kappa(\mathcal{I}_1)}$ is $\mathtt{T}$, but $\phi_{\kappa(\mathcal{I}_1)} \not\rightarrow \phi_{\kappa(\mathcal{I}_2)}$. We conclude this section by illustrating how the two cases above are mapped into the A/G contract framework, using the following two examples.

Example 14 (Feedback-Induced Inconsistency)**.** *Consider $\mathcal{I}_A = (\{x, z\}, \{y\}, \mathtt{T})$ and $\mathcal{I}_B = (\{x, z\}, \{y\}, x \neq y)$ as in Figure 3.2 (a). $\mathcal{I}_A$ does not make any assumptions on the inputs and any guarantee on the outputs, while $\mathcal{I}_B$ guarantees that the value of the output is different from the value of the input. We have $\mathcal{I}_B \sqsubseteq \mathcal{I}_A$ since $in(\phi_A) = in(\phi_B) = \mathtt{T}$ and $\phi_B \rightarrow \phi_A$. However, given $\kappa(\mathcal{I}_A) = (\{z\}, \{y, x\}, x = y)$ and $\kappa(\mathcal{I}_B) = (\{z\}, \{y, x\}, \mathtt{F})$, obtained as shown in Figure 3.2 (b), clearly $\kappa(\mathcal{I}_B) \not\sqsubseteq \kappa(\mathcal{I}_A)$ since $\phi_{\kappa(\mathcal{I}_B)}$ is $\mathtt{F}$.*

Consider now the associated contracts $A = (V, \mathtt{T}, \mathtt{T})$ and $B = (V, \mathtt{T}, \Box(y \neq x))$ on variables $V = \{x, y, z\}$. We have $B \preceq A$, $\kappa(A) = (V, \mathtt{T}, \Box(y = x))$, $\kappa(B) = (V, \mathtt{T}, \mathtt{F})$, and $\kappa(B) \preceq \kappa(A)$. Therefore, refinement is preserved by feedback composition. However, the feedback connection has caused an inconsistency *internal to $\kappa(B)$; $\phi_{\kappa(\mathcal{I}_B)} = \mathtt{F}$ reflects $\kappa(B)$ being* inconsistent*.*

Example 15 (Feedback-Induced Incompatibility)**.** *Consider the two interfaces $\mathcal{I}_A$ and $\mathcal{I}_B$ defined as follows:*

$$\mathcal{I}_A = (\{x\}, \{y\}, (x \neq 0) \wedge (xy = 1)),$$
$$\mathcal{I}_B = (\{x\}, \{y\}, (x \neq 0) \rightarrow (xy = 1)).$$

We have $\mathcal{I}_B \sqsubseteq \mathcal{I}_A$ *since* $in(\phi_A) = (x \neq 0)$, $in(\phi_B) = \mathtt{T}$ *and* $(\phi_B \to \phi_A) = (x \neq 0)$. *However, given* $\kappa(\mathcal{I}_A)$ *and* $\kappa(\mathcal{I}_B)$ *computed as follows:*

$$\kappa(\mathcal{I}_A) = (\emptyset, \{y, x\}, (x^2 = 1) \wedge (x = y)),$$

$$\kappa(\mathcal{I}_B) = (\emptyset, \{y, x\}, (x \neq 0 \to x^2 = 1) \wedge (x = y)),$$

we obtain $in(\phi_{\kappa(\mathcal{I}_B)}) = in(\phi_{\kappa(\mathcal{I}_A)}) = \mathtt{T}$, $\phi_{\kappa(\mathcal{I}_B)} \not\to \phi_{\kappa(\mathcal{I}_A)}$, *and therefore* $\kappa(\mathcal{I}_B) \not\sqsubseteq \kappa(\mathcal{I}_A)$. *In fact,* $\kappa(\mathcal{I}_B)$ *admits all sequences of the form* (x_n, x_n), *with* $x_n \in \{-1, 0, 1\}$ *for all* $n \in \mathbb{N}$, *while* $x_n = 0$ *is not allowed by* $\kappa(\mathcal{I}_A)$. *Consider now the associated contracts* A *and* B *defined below on variables* $V = \{x, y\}$*:*

$$A = (V, \Box(x \neq 0), \Box(x \neq 0) \to \Box(xy = 1)),$$

$$B = (V, \mathtt{T}, \Box((x \neq 0) \to (xy = 1)).$$

We can compute $\kappa(A)$ *and* $\kappa(B)$ *as*

$$\kappa(A) = (V, \Box(x \neq 0) \vee \neg\Box(y = x), (\Box(x \neq 0) \to \Box(x^2 = 1)) \wedge \Box(y = x)),$$

$$\kappa(B) = (V, \mathtt{T}, \Box((x \neq 0) \to (x^2 = 1)) \wedge (y = x)).$$

In this case, $\kappa(B)$ *is* compatible *and* consistent. *Moreover, we have* $\kappa(B) \preceq \kappa(A)$, *i.e. refinement is preserved by feedback. However, even if* $\kappa(A)$ *is compatible for the general A/G contract framework, the assumption projection of* $\kappa(A)$ *with respect to either* x *or* y *unveils a potential source of* incompatibility *induced by feedback. For example, the assumptions of* $\mathrm{AP}_y(\kappa(A))$ *would require* $\Box x \neq 0$. *However, such a condition may not be guaranteed* per se *by the feedback connection.*

3.2.7 Conjunction

In this section, we will use the term conjunction to also denote shared refinement, which is the counterpart of conjunction for relational interfaces. We show that even if $\mathcal{F}$ preserves refinement, it does not preserve conjunction. First, as mentioned in Section 3.2.1, conjunction cannot always be defined for relational interfaces. For A/G contracts, conjunction is instead always defined as the GLB of the refinement relation. However, when the set of shared refinements of two interfaces is empty, the conjunction of their associated contracts can become inconsistent, as illustrated by the following example.

Example 16 (Inconsistent Conjunction)**.** *Consider* $\mathcal{I}_{00} = (\{x\}, \{y\}, x = 0 \to y = 0)$ *and* $\mathcal{I}_{01} = (\{x\}, \{y\}, x = 0 \to y = 1)$. *They are not* shared refinable*, since it is not possible to guarantee both* $y = 0$ *and* $y = 1$ *when* $x = 0$ *[197]. However, conjunction can still be defined for the associated contracts* $\mathcal{F}(\mathcal{I}_{00}) = (\{x, y\}, \mathtt{T}, \Box(x = 0 \to y = 0))$ *and* $\mathcal{F}(\mathcal{I}_{01}) = (\{x, y\}, \mathtt{T}, \Box(x = 0 \to y = 1))$*, although it corresponds to the inconsistent contract* $(\{x, y\}, \mathtt{T}, \mathtt{F})$*, i.e. the "bottom" element for the refinement preorder.*

When conjunction is well-defined in relational interfaces, the contract associated with the conjunction of two interfaces is, in general, a refinement of the conjunction of the contracts associated with the interfaces, as stated by the following theorem.

Theorem 3.2.9 (Interface Conjunction Refines Contract Conjunction)**.** *Let* $\mathcal{I} = (X, Y, \phi)$ *and* $\mathcal{I}' = (X, Y, \phi')$ *be two* shared-refinable *relational interfaces. Then we have*

$$\mathcal{F}(\mathcal{I} \sqcap \mathcal{I}') \preceq \mathcal{F}(\mathcal{I}) \wedge \mathcal{F}(\mathcal{I}'), \tag{3.13}$$

where, in general, $\mathcal{F}(\mathcal{I} \sqcap \mathcal{I}') \neq \mathcal{F}(\mathcal{I}) \wedge \mathcal{F}(\mathcal{I}')$.

While (3.13) can be easily proved as a direct consequence of Theorem 3.2.5, our main goal is to gather insight into the reason why the equality does not generally hold, and the GLB is not preserved by $\mathcal{F}$. Therefore, we present an alternative proof, which directly reasons about the assumption and guarantee sets of the contracts in (3.13).

Proof (Theorem 3.2.9). We recall that $\mathcal{I} \sqcap \mathcal{I}' = (X, Y, \phi_\sqcap)$, where

$$\phi_\sqcap = (in(\phi) \vee in(\phi')) \wedge (in(\phi) \rightarrow \phi) \wedge (in(\phi') \rightarrow \phi'),$$

and $in(\phi_\sqcap) = in(\phi) \vee in(\phi')$ by Lemma 8 in [197]. Therefore, by transforming $\mathcal{I} \sqcap \mathcal{I}'$, we obtain $\mathcal{F}(\mathcal{I} \sqcap \mathcal{I}') = (X \cup Y, A_\sqcap, G_\sqcap)$, where $A_\sqcap = \Box(in(\phi) \vee in(\phi'))$ and

$$G_\sqcap = \Box(in(\phi) \vee in(\phi')) \rightarrow (\Box(in(\phi) \rightarrow \phi) \wedge \Box(in(\phi') \rightarrow \phi')).$$

Moreover, by definition of conjunction, we obtain $\mathcal{F}(\mathcal{I}) \wedge \mathcal{F}(\mathcal{I}') = (X \cup Y, A_\wedge, G_\wedge)$, where $A_\wedge = \Box in(\phi) \vee \Box in(\phi')$ and

$$G_\wedge = (\Box in(\phi) \rightarrow \Box\phi) \wedge (\Box in(\phi') \rightarrow \Box\phi').$$

It is easy to see that $A_\wedge \rightarrow A_\sqcap$. On the other hand, we also notice that $A_\sqcap \not\rightarrow A_\wedge$ in general. In fact, any sequence x_n such that $x_1 \models in(\phi)$, $x_1 \not\models in(\phi')$, and $x_n \models in(\phi')$ for all $n > 1$, satisfies $A_\sqcap$ but does not satisfy $A_\wedge$.

We also observe that $G_\sqcap \rightarrow G_\wedge$. In fact, $G_\wedge$ is trivially `T` if both $\Box in(\phi)$ and $\Box in(\phi')$ are `F`. If $\Box in(\phi)$ is instead `T`, then, because $G_\sqcap$ is `T`, $\Box(in(\phi) \rightarrow \phi)$ is `T`, which implies that $\Box\phi$ is also `T`. Similarly, if $\Box in(\phi')$ is `T`, $\Box\phi'$ will also be `T`. Therefore, in all cases, both the implications in $G_\wedge$ will be `T` under the assumption that $G_\sqcap$ is `T`. On the other hand, we also notice that $G_\wedge \not\rightarrow G_\sqcap$ in general. In fact, any sequence (x_n, y_n) such that $x_1 \models in(\phi)$, $x_1 \not\models in(\phi')$, $x_n \models in(\phi')$ for all $n > 1$, and $(x_1, y_1) \not\models \phi$, would certainly satisfy $G_\wedge$ but not $G_\sqcap$. □

As stated by Theorem 3.2.9, the contract associated with the conjunction of $\mathcal{I}$ and $\mathcal{I}'$ is not generally equal to the conjunction of the contracts associated with $\mathcal{I}$ and $\mathcal{I}'$. Moreover, the expressions of the assumptions and guarantees computed in our proof highlight another crucial difference between the two frameworks, which explains this result. While an

A/G contract reasons about the entire behavior of a component, possibly spanning infinite sequences of reactions, a relational interface can constrain the inputs and outputs of a component at the granularity of a single reaction index. Therefore, computation of conjunction generates a smaller set of allowed environments and a larger set of guaranteed behaviors for A/G contracts, which translates into a tighter, less conservative, greatest lower bound.

3.3 Compatibility and Consistency in A/G Contracts

Based on the results of Section 3.2, we can now complete our discussion of compatibility and consistency of contracts that make a distinction between input (uncontrolled) and output (controlled) variables.

Let $U \subseteq V$ and $Y \subseteq V$ be, respectively, the subset of input and output variables of a contract $\mathcal{C}$, with $U \cap Y = \emptyset$. Moreover, to be concrete, we assume that the assumptions and guarantees of $\mathcal{C}$ are represented in terms of predicates or logic formulas on their variables, i.e. $\mathcal{C} = (U \cup Y, \phi_A, \phi_G)$. Then, checking that $\mathcal{C}$ is *compatible* by the definition in Section 2.3.3 translates into checking that A is *Y-receptive*. In our case, this is equivalent to verifying that $(\forall Y : \phi_A)$ is satisfiable. We observe that we can conveniently restate this compatibility condition using the assumption-projection operator in Definition 3.2.2, by requiring that the $\mathrm{AP}_Y(\mathcal{C})$ is compatible. Similarly, $\mathcal{C}$ is *consistent* if and only if G is *X-receptive*, i.e., for contracts in saturated form, that $(\forall X : \phi_G) \neq \mathtt{F}$.

The definitions above can be lifted to pairs of contracts, so that $\mathcal{C}_1$ and $\mathcal{C}_2$ are compatible (consistent) if and only if $\mathcal{C}_1 \otimes \mathcal{C}_2$ is compatible (consistent). In particular, we can reconsider compatibility and consistency conditions for the cascade of contracts in Figure 2.4 (a), where $\mathcal{C}_1 = (U_1 \cup Y_1, \phi_{A1}, \phi_{G1})$ and $\mathcal{C}_2 = (U_2 \setminus U_2^\sigma \cup Y_1^\sigma \cup Y_2, \phi_{A2}, \phi_{G2})$. We assume that the original contracts $\mathcal{C}_1$ and $\mathcal{C}_2$ are themselves compatible and consistent. In fact, if any contract is incompatible, then it cannot be used in any context; if any contract is inconsistent, it translates into a specification that generates *per se* a contradiction, and cannot be implemented. Assumptions and guarantees for the composite contract $\mathcal{C}_\sigma = \mathcal{C}_1 \overset{\sigma}{\rightsquigarrow} \mathcal{C}_2$ have been computed in (2.5) and (2.6), where $\phi_{G\sigma}$ and $\phi_{A\sigma}$ are interpreted as predicates or formulas over the entire set of variables $U_1 \cup Y_1 \cup U_2 \setminus U_2^\sigma \cup Y_2$. Finally, since $\mathcal{C}_1$ and $\mathcal{C}_2$ are in saturated form, we recall that both $\phi_{A1} \vee \phi_{G1} = \mathtt{T}$ and $\phi_{A2} \vee \phi_{G2} = \mathtt{T}$ must hold.

When there is a distinction between controlled output and uncontrolled input variables, we can assume that each component only controls its outputs, while the inputs are assigned by the external environment. In this scenario, the assumptions of each contract will only involve its input variables, since the outputs will be under the responsibility of the implementations. Specifically, for the system in Figure 2.4 (a), ϕ_{A1} and ϕ_{A2} will depend, respectively, only on U_1 and $Y_1^\sigma \cup U_2 \setminus U_2^\sigma$, while ϕ_{G1} and ϕ_{G2} will describe relations involving both the input and output variables of each contract. Our objective is to determine the conditions on the input variables $U_1 \cup U_2 \setminus U_2^\sigma$ that make the composite contract $\mathcal{C}_\sigma$ compatible. To do so, since the variables in $Y_1 \cup Y_2$ cannot be controlled by the environment, we derive new assumptions for $\mathcal{C}_\sigma$ by assumption-projection, i.e. by using universal quantification over the

output variables $Y = Y_1 \cup Y_2$ as follows:

$$\begin{aligned}\phi'_{A\sigma} &:= \forall Y : \phi_{A\sigma} \\ &= \neg\exists Y : (\neg\phi_{A1} \wedge \phi_{G1} \wedge \phi_{G2}) \vee (\neg\phi_{A2} \wedge \phi_{G1} \wedge \phi_{G2}) \\ &= (\neg\exists Y : \neg\phi_{A1} \wedge \phi_{G1} \wedge \phi_{G2}) \wedge (\neg\exists Y : \neg\phi_{A2} \wedge \phi_{G1} \wedge \phi_{G2}) \\ &= (\neg\exists Y : \neg\phi_{A1} \wedge \phi_{G2}) \wedge (\neg\exists Y : \neg\phi_{A2} \wedge \phi_{G1}) \\ &= (\phi_{A1} \vee (\forall Y : \neg\phi_{G2})) \wedge (\forall Y : \phi_{G1} \rightarrow \phi_{A2}). \end{aligned} \tag{3.14}$$

In the derivations above, we leverage the fact that $\neg\phi_{A1} \rightarrow \phi_{G1}$ and $\neg\phi_{A2} \rightarrow \phi_{G2}$ must always hold for contracts in saturated form (thus implying, e.g., $\neg\phi_{A1} \wedge \phi_{G1} \wedge \phi_{G2} = \neg\phi_{A1} \wedge \phi_{G2}$). Furthermore, since the contrapositive $\neg\phi_{G2} \rightarrow \phi_{A2}$ is also true for $\mathcal{C}_2^\sigma$ (implying $\neg\phi_{G2} = \neg\phi_{G2} \wedge \phi_{A2}$), we obtain

$$(\forall Y : \neg\phi_{G2}) = (\forall Y : \neg\phi_{G2} \wedge \phi_{A2}) = \mathtt{F}, \tag{3.15}$$

since for any input set satisfying ϕ_{A2}, there always exists a set of outputs Y_2 that satisfies ϕ_{G2}. Given that ϕ_{A1} does not depend on Y, we can then conclude from (3.14) that

$$\phi'_{A\sigma} = \phi_{A1} \wedge (\forall Y : \phi_{G1} \rightarrow \phi_{A2}), \tag{3.16}$$

and that $\mathcal{C}_\sigma$ is compatible if and only if $\phi'_{A\sigma}$ is satisfiable. Intuitively, this is equivalent to require that there exists an environment satisfying the assumptions of $\mathcal{C}_1$, and capable of driving the guarantees of $\mathcal{C}_1$ to become a subset of the assumptions of $\mathcal{C}_2^\sigma$ for any possible assignment of the output variables. For example, we apply the result above to the serial composition $\mathcal{C}_{sin} \overset{\sigma}{\rightsquigarrow} \mathcal{C}_{amp}$ in Figure 2.3 (b).

Example 17 (Compatibility and Serial Composition)**.** *Both $\mathcal{C}_{sin}$ and $\mathcal{C}_{amp}$ are compatible and consistent contracts. Moreover, $\phi_{A,sin} = \mathtt{T}$. However, for their composition to be compatible, we also need to enforce $(\forall Y : \phi_{G1} \rightarrow \phi_{A2})$:*

$$\begin{aligned}\forall x : \forall y : (x \neq 2\sin\theta) \vee (|x| \leq 1) = \\ &= \neg\exists x : (x = 2\sin\theta) \wedge (|x| > 1) \\ &= \neg\exists x : (x = 2\sin\theta) \wedge (2|\sin\theta| > 1) \\ &= \neg\left((2|\sin\theta| > 1) \wedge (\exists x : x = 2\sin\theta)\right) \\ &= |\sin\theta| \leq \frac{1}{2} \vee \neg\exists x : (x = 2\sin\theta) \\ &= |\sin\theta| \leq \frac{1}{2} \\ &\iff \bigvee_{k\in\mathbb{Z}} \left(-\frac{\pi}{6} + k\pi \leq \theta \leq \frac{\pi}{6} + k\pi\right). \end{aligned}$$

In fact, if θ violates this condition, there is no way for $\mathcal{C}_{sin}$ to provide a legal environment for $\mathcal{C}_{amp}^\sigma$.

Compatibility and consistency conditions for the feedback composition in Figure 2.4 (b) can be determined in a similar way. Computation of the composite contract $\mathcal{C}_\kappa = \mathcal{C}_1^\kappa \circ_\kappa \mathcal{C}_2^\kappa$ generates expressions for the assumptions $\phi_{A\kappa}$ and the guarantees $\phi_{G\kappa}$ that are analogous to (2.5) and (2.6). However, in the special case of controlled outputs and uncontrolled inputs, we obtain

$$\phi'_{A\kappa} := \forall Y : \phi_{A\kappa} = \forall Y : (\phi_{G2} \to \phi_{A1}) \wedge (\phi_{G1} \to \phi_{A2}), \tag{3.17}$$

stating that $\mathcal{C}_\kappa$ is compatible if and only if there exists an environment such that, for all possible assignments on the output variables, the guarantees of $\mathcal{C}_1^\kappa$ are included into the assumptions of $\mathcal{C}_2^\kappa$, and *vice versa*.

Example 18 (Compatibility and Feedback Composition)**.** *We investigate the feedback composition of the* Square *component with the* Diode *component in Figure 2.3 (c), where* $\mathcal{C}_{square} = (\{w, z\}, \mathtt{T}, z = w^2)$ *and* $\mathcal{C}_{diode} = (\{w, z\}, z \geq 0, (z < 0) \vee (w = z))$*. Then, since*

$$\forall w : \forall z : ((z < 0) \vee (w = z) \to \mathtt{T}) \wedge (z = w^2 \to z \geq 0) = \mathtt{T},$$

we conclude that the two contracts are compatible, which is consistent with the results computed in Example 6 for the assumptions of the composite contracts. Moreover, the admitted behaviors can be obtained from the joint guarantees $\phi_{G,square} \wedge \phi_{G,diode} := (w = z) \wedge (z = 0 \vee z = 1)$*.*

Finally, for a contract $\kappa(\mathcal{C}) = (U \cup Y \setminus y, \phi_A, \phi_G)$ as in Figure 2.4 (c), compatibility checking reduces to verifying that

$$\phi'_A := \forall Y : \phi_A = \forall Y : \phi_A \vee \neg\phi_G = \forall Y : \phi_G \to \phi_A \tag{3.18}$$

is satisfiable, where we use again the fact that $\neg\phi_G \to \phi_A$ is always true for a contract in saturated form.

Special cases of (3.16), (3.17), and (3.18) have been previously reported in the literature, under some restrictive conditions that contract assumptions and guarantees are attached, respectively, at each input and output port of a component independently, and they can be expressed uniquely as functions of the attached ports [36]. By embedding compatibility and consistency within the generic A/G contract framework, we can instead provide more general results.

3.4 Heterogeneous Refinement and Vertical Contracts

A key challenge in CPS design is relating the semantics of different models, which are attached to the same underlying system, but are defined in different formalisms. To create a formal framework that can work with every formalism, while being at the same time independent of the specifics of any of them, we rely on the fact that A/G contract are defined out of sets of behaviors. Therefore, as a first step, we use *behavior mappings* to introduce a

new relation, *heterogeneous refinement*, which extends the classical relation of refinement to contracts expressed using different formalisms.

Heterogeneous refinement is, however, not enough to express refinement between the specification contract and the implementation contract, when they are attached to heterogeneous modeling architectures, and present heterogeneous structures. Therefore, as a second step, we also equip our framework with a new notion of composition that can relate a contract and its vertical (heterogeneous) refinement, including their different viewpoints, independently of their modeling structures. Checking compatibility and consistency of the resulting *vertical contract* becomes then instrumental to assess the overall design correctness.

3.4.1 Heterogeneous Refinement

The notion of refinement introduced in Section 2.3.4 can be generalized to the case of two contracts, $\mathcal{C}_1$ and $\mathcal{C}_2$, which are also expressed by using different formalisms. In this case, before a refinement relation can be defined, we need to map the behaviors expressed by one of the contracts to the domain of the other contract via a transformation $\mathcal{M}$ (e.g. a special type of projection or inverse projection) which is generally more involved than alphabet equalization.

Let B_1 and B_2 be two sets of behaviors, possibly defined in the different formalisms $\mathcal{B}_1$ and $\mathcal{B}_2$, respectively. Behavior formalisms may include, for instance, event traces, continuous signals, or hybrid trajectories. We define mappings between different behavior domains in terms of *abstraction functions* as follows.

Definition 3.4.1 (Behavior Abstraction Function)**.** *Given two behavior domains B_1 and B_2 in possibly different behavior formalisms $\mathcal{B}_1$ and $\mathcal{B}_2$, a* behavior abstraction function *is a function $\mathcal{M} : B_1 \rightarrow B_2$ that associates each behavior $\beta_1 \in B_1$ with one and only one behavior $\beta_2 = \mathcal{M}(\beta_1) \in B_2$.*

Abstraction functions are often problem-specific and they are usually assumed informally any time two different models M_1 and M_2 of the same system are created with behaviors $[\![M_1]\!]$ and $[\![M_2]\!]$ expressed in different formalisms $\mathcal{B}_1$ and $\mathcal{B}_2$; our framework aims to facilitate the explicit and rigorous definition of abstraction functions as the models get created.

Let us assume that behaviors in B_1 and B_2 are defined, respectively, over the sets of variables V_1 and V_2. Then, while mapping behaviors in B_1 to behaviors in B_2, $\mathcal{M}$ will also establish a mapping between the variable sets V_1 and V_2; this mapping will be, in general, a relation $R_{\mathcal{M}} \subseteq V_1 \times V_2$. In the following, we use the notation $\mathcal{M}(B_1')$ to denote the image of a behavior set $B_1' \subseteq B_1$ via the mapping $\mathcal{M}$, i.e. $\mathcal{M}(B_1') = \{b_2 | b_2 = \mathcal{M}(b_1), \forall b_1 \in B_1'\}$. Furthermore, for a subset of variables $V_1' \subseteq V_1$, we use the notation $R_{\mathcal{M}}(V_1')$ to denote the subset of variables in V_2 associated with the variables in V_1', i.e. $R_{\mathcal{M}}(V_1') = \{v_2 \in V_2 | \exists v_1 \in V_1' : (v_1, v_2) \in R_{\mathcal{M}}\}$. Similarly, we denote the inverse image of a behavior set $B_2' \subseteq B_2$ via the mapping $\mathcal{M}$ as $\mathcal{M}^{-1}(B_2')$, i.e. $\mathcal{M}^{-1}(B_2') = \{b_1 | \mathcal{M}(b_1) \in B_2'\}$, and the subset of variables in V_1 associated with the set of variables $V_2' \subseteq V_2$ as $R_{\mathcal{M}}^{-1}(V_2')$, i.e. $R_{\mathcal{M}}^{-1}(V_2') = \{v_1 \in V_1 | \exists v_2 \in V_2' : (v_1, v_2) \in R_{\mathcal{M}}\}$.

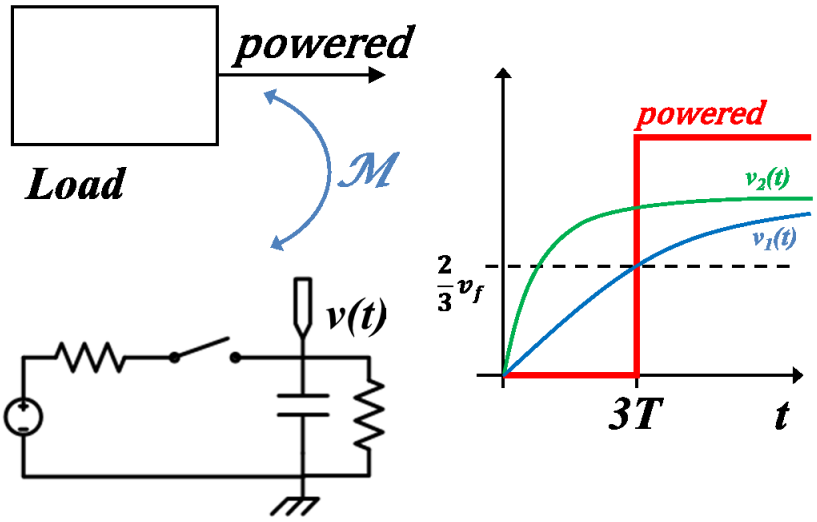


Figure 3.3: Example of heterogeneous refinement.

Based on the definitions above, we can then introduce the notion of heterogeneous refinement.

Definition 3.4.2 (Heterogeneous Refinement)**.** *Let B_1 and B_2 be two behavior domains, including, respectively, behaviors over the variable sets V_1 and V_2, and possibly expressed using different formalisms $\mathcal{B}_1$ and $\mathcal{B}_2$; let $\mathcal{M}$ be a behavior abstraction function from B_1 to B_2. Given contracts $\mathcal{C}_1 = (V_1, A_1, G_1)$ and $\mathcal{C}_2 = (V_2, A_2, G_2)$, both in saturated form, and such that $A_1, G_1 \subseteq B_1$, $A_2, G_2 \subseteq B_2$, and $V_1 = R_{\mathcal{M}}^{-1}(V_2)$, we say that $\mathcal{C}_1$* refines *$\mathcal{C}_2$* via *$\mathcal{M}$, written $\mathcal{C}_1 \preceq_{\mathcal{M}} \mathcal{C}_2$, if and only if $A_1 \supseteq \mathcal{M}^{-1}(A_2)$ and $G_1 \subseteq \mathcal{M}^{-1}(G_2)$.*

Example 19 (Heterogeneous Refinement)**.** *Let $\mathcal{C}_{dis} = (\{powered\}, \mathtt{T}, \Diamond_{[0,3)} powered)$ be the contract specifying the dynamics of a load in an electrical system, which is powered at startup. $\mathcal{C}_{dis}$ offers a discrete-time discrete-state abstraction of the dynamics, prescribing that, in all contexts, the Boolean variable powered must be asserted within three time units. On the other hand, let $\mathcal{C}_{con} = (\{v\}, \mathtt{T}, v(t) = v_f(1 - \mathrm{e}^{-\frac{t}{\tau}}), t \in \mathbb{R}, t \geq 0)$ be the contract describing the voltage level of the electrical load as a continuous function of time t. The load responds as a first order dynamical system with time constant τ and steady-state voltage v_f. Then, we can reason about refinement between $\mathcal{C}_{con}$ and $\mathcal{C}_{dis}$ only if we provide a mechanism to map continuous time and voltage levels into discrete ones. In this case, given a time step T, such*

a mechanism could be provided by the following transformation $\mathcal{M}$:

$$\mathcal{M} : \left\{ \begin{array}{rl} powered & := (v \geq \frac{2}{3} v_f) \\ k & := \lfloor \frac{t}{T} \rfloor \end{array} \right. , \tag{3.19}$$

stating that powered is asserted if and only if the voltage is greater than or equal to two thirds of the steady-state value, while the discrete time index k is obtained by discretizing t according to the quantization step T. Resting on this mapping, we can then conclude that $\mathcal{C}_{con} \preceq_{\mathcal{M}} \mathcal{C}_{dis}$ if and only if $v(3T) > \frac{2}{3} v_f$, i.e. if and only if the system time constant satisfies $\tau < \frac{3T}{\ln 3}$. This condition is illustrated in Figure 3.3, where $v_2(t)$ (in green) satisfies the constraint on τ and refines the guarantees of $\mathcal{C}_{dis}$, whereas $v_1(t)$ (in blue) does not, since it reaches the desired value $\frac{2}{3} v_f$ exactly at time $t = 3T$ ($k = 3$), while the interval in the guarantees of $\mathcal{C}_{dis}$ is right-open.

As shown in Example 19, heterogeneous refinement allows checking refinement between contracts in different formalisms. Finally, it is also possible to show that heterogeneous refinement is preserved by composition, as stated by the following proposition.

Proposition 3.4.3 (Compositional Heterogeneous Refinement)**.** *Let B and B' be behavior domains over the variable sets V and V', expressed in formalisms $\mathcal{B}$ and $\mathcal{B}'$, respectively; let $\mathcal{M} : B \to B'$ be a behavior abstraction function, with $B' = \mathcal{M}(B)$ and $V = R^{-1}_{\mathcal{M}}(V')$. Let $\mathcal{C}_1, \mathcal{C}_2$ be A/G contracts defined in the behavior domain B, and $\mathcal{C}'_1, \mathcal{C}'_2$ be A/G contracts defined in the domain B', all in saturated form. If $\mathcal{C}_1 \preceq_{\mathcal{M}} \mathcal{C}'_1$, $\mathcal{C}_2 \preceq_{\mathcal{M}} \mathcal{C}'_2$, and $\mathcal{C}'_1$ is compatible with $\mathcal{C}'_2$, then $\mathcal{C}_1$ is also compatible with $\mathcal{C}_2$ and $\mathcal{C}_1 \otimes \mathcal{C}_2 \preceq_{\mathcal{M}} \mathcal{C}'_1 \otimes \mathcal{C}'_2$.*

Proof. Let $\mathcal{C}'_1 = (V', A'_1, G'_1)$ and $\mathcal{C}'_2 = (V', A'_2, G'_2)$, with $A'_1, G'_1, A'_2, G'_2 \subseteq B'$, and $\mathcal{C}_1 = (V, A_1, G_1)$ and $\mathcal{C}_2 = (V, A_2, G_2)$, with $A_1, G_1, A_2, G_2 \subseteq B$. By hypothesis, we have $G_1 \subseteq \mathcal{M}^{-1}(G'_1)$, $G_2 \subseteq \mathcal{M}^{-1}(G'_2)$, $A_1 \supseteq \mathcal{M}^{-1}(A'_1)$, $A_2 \supseteq \mathcal{M}^{-1}(A'_2)$. Therefore, for the guarantees of $\mathcal{C}_1 \otimes \mathcal{C}_2$ we obtain

$$G_{12} = G_1 \cap G_2 \subseteq \mathcal{M}^{-1}(G'_1) \cap \mathcal{M}^{-1}(G'_2) = \mathcal{M}^{-1}(G'_1 \cap G'_2) = \mathcal{M}^{-1}(G'_{12}). \tag{3.20}$$

On the other hand, for the assumptions of $\mathcal{C}'_1 \otimes \mathcal{C}'_2$ we obtain

$$\begin{aligned} \mathcal{M}^{-1}(A'_{12}) = \mathcal{M}^{-1}\left((A'_1 \cap A'_2) \cup \overline{G'_1 \cap G'_2}\right) &= \left(\mathcal{M}^{-1}(A'_1) \cap \mathcal{M}^{-1}(A'_2)\right) \cup \mathcal{M}^{-1}(\overline{G'_1 \cap G'_2}) \\ &= \left(\mathcal{M}^{-1}(A'_1) \cap \mathcal{M}^{-1}(A'_2)\right) \cup \overline{\mathcal{M}^{-1}(G'_1 \cap G'_2)} \\ &\subseteq (A_1 \cap A_2) \cup \overline{G_1 \cap G_2} = A_{12}, \end{aligned} \tag{3.21}$$

where we use (3.20) in the last step of (3.21)[4]. By definition of heterogeneous refinement, (3.20) and (3.21) allow us to conclude $\mathcal{C}_1 \otimes \mathcal{C}_2 \preceq_{\mathcal{M}} \mathcal{C}'_1 \otimes \mathcal{C}'_2$. Moreover, by compatibility of $\mathcal{C}'_1$ and $\mathcal{C}'_2$, and by B' being the image of B under $\mathcal{M}$, we have that $\mathcal{M}^{-1}(A'_{12})$ is not empty, hence A_{12} is not empty, which means that $\mathcal{C}_1$ is also compatible with $\mathcal{C}_2$. □

[4]We also use the fact that $\mathcal{M}^{-1}(\overline{A'}) = \overline{\mathcal{M}^{-1}(A')}$ for any subset A' of the universal set B'. In fact, we have $B = \mathcal{M}^{-1}(B') = \mathcal{M}^{-1}(A' \cup \overline{A'}) = \mathcal{M}^{-1}(A') \cup \mathcal{M}^{-1}(\overline{A'})$, $\emptyset = \mathcal{M}^{-1}(A' \cap \overline{A'}) = \mathcal{M}^{-1}(A') \cap \mathcal{M}^{-1}(\overline{A'})$, which jointly lead to $\mathcal{M}^{-1}(\overline{A'}) = \overline{\mathcal{M}^{-1}(A')}$.

Therefore, compatible contracts can be independently refined, which is key to enable top-down incremental design with heterogeneous formalisms, by iteratively decomposing a system-level contract $\mathcal{C}$ in formalism $\mathcal{B}'$ into sub-system contracts $\mathcal{C}_i$ in formalisms $\mathcal{B}$ for further independent development.

3.4.2 Vertical Contracts

Traditionally contracts have been used to specify components, and aggregation of components at the *same level of abstraction*, as in the A/G framework presented in Section 2.3; for this reason we refer to them as *horizontal contracts*.

We use contracts also to formalize and reason about refinement between two different abstraction levels in the PBD process [152, 34]; for this reason, we refer to this type of contracts as *vertical contracts*. To illustrate this concept, consider the problem of mapping a specification platform of a system at level $l+1$ into an implementation platform at level l. In general, the specification platform architecture (i.e. interconnection of components) may be defined in an independent way, and may not directly match the implementation platform architecture. Such a *heterogeneous architectural decomposition* will also reflect on the contracts associated with the components and their aggregations. For instance, the contract describing the specification platform $\mathcal{C} = \bigwedge_{k\in K} \left(\bigotimes_{i\in I_k} \mathcal{C}_{ik}\right)$ may be defined as the conjunction of K different viewpoints, each characterized by its own architectural decomposition into I_k contracts. On the other hand, the contract describing the implementation platform $\mathcal{M} = \bigotimes_{j\in J} \left(\bigwedge_{n\in N_j} \mathcal{M}_{jn}\right)$ may be better represented as a composition of J contracts, each defined out of a conjunction of its different viewpoints. Because there may not be, in general, a direct matching between contracts and viewpoints of $\mathcal{M}$ and $\mathcal{C}$, checking that $\mathcal{M} \preceq \mathcal{C}$ in a compositional way, by reasoning on the elements of $\mathcal{M}$ and $\mathcal{C}$ independently, as discussed in Section 2.3.4 (for classical refinement) and Section 3.4.1 (for heterogeneous refinement), may not be effective.

However, it is still possible to reason about refinement between $\mathcal{M}$ and $\mathcal{C}$ by resorting to a contract which specifies the composition of a model and its vertical refinement, even though they are not directly connected, by connecting them indirectly through a mapping, e.g., by synchronizing pairs of events, as if co-simulating a model and its refinement. Informally, this kind of composition captures the fact that the actual satisfaction of all the design requirements and viewpoints by a deployment depends on the supporting execution platform, the underlying physical system, and on the way in which system functionalities are mapped into them. Formally, this composition can be modelled using two alternative methods, based on the specific shapes of $\mathcal{C}$ and $\mathcal{M}$:

- The interaction between the specification and the implementation platforms can be modeled using the contract composition $\mathcal{C} \otimes \mathcal{M}$. In this case, assumptions made by the specification platform on the implementation platform can be discharged by the guarantees of the implementation platform, and *vice versa*, as indicated by (2.2) and (2.3). Refinement can then be checked by checking that $\mathcal{C} \otimes \mathcal{M}$ is compatible and

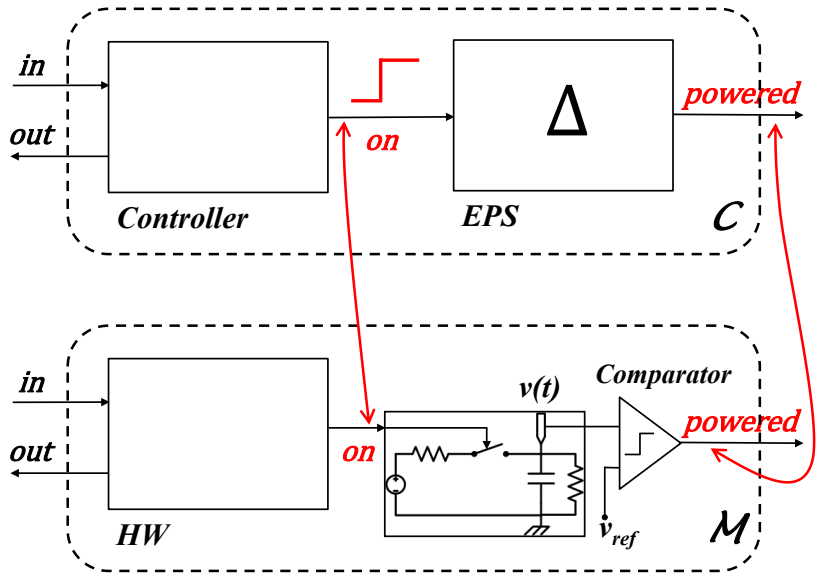


Figure 3.4: Specification and implementation platform examples used to illustrate vertical contracts.

that $\mathcal{C} \otimes \mathcal{M} \preceq \mathcal{C}$, which can be performed compositionally, by matching the elements of $\mathcal{C}$ with the ones of $\mathcal{C} \otimes \mathcal{M}$.

- The interaction between the specification and the implementation platforms can also be modeled using the contract conjunction $\mathcal{C} \wedge \mathcal{M}$. In this case, assumptions and guarantees combine as in (2.12), and $\mathcal{C} \wedge \mathcal{M}$ is assured to refine $\mathcal{C}$ by construction. However, being a conjunction, it can still be a source of inconsistencies. Therefore, to guarantee that the design can be implemented, the consistency of $\mathcal{C} \wedge \mathcal{M}$ must be checked or enforced by the designer.

Composite contracts such as $\mathcal{C} \otimes \mathcal{M}$ and $\mathcal{C} \wedge \mathcal{M}$ are both called *vertical contracts*, and can be used to formalize mechanisms for mapping a specification over an execution platform, such as the ones adopted in the RT-Builder tool[5], in the METROPOLIS [24] and METROII [25] frameworks, and, more recently, the METRONOMY framework [90], which provides support for timing contract verification. We refer the reader to the literature on timing analysis of embedded software, e.g. [201, 182, 183], for further references on timing verification. We exemplify below the use of vertical contracts by referring to the virtual model of a simple system pictured in Figure 3.4.

[5] `http://www.geensoft.com/en/article/rtbuilder`

Example 20 (Vertical Contracts as Parallel Composition)**.** *The specification platform architecture at the top of Figure 3.4 consists of two interconnected components. At startup, the* Controller *interacts with an external subsystem through its* in *and* out *ports to perform some high-priority task. Then, it switches on a safety-critical electric power system* EPS*, by asserting its output* on*, and makes sure that the system is actually powered, i.e. the signal* powered *is asserted, by the deadline* t_d*.*

At the application level, to conveniently explore different control strategies, the designer abstracts the physical system EPS *using a simple delay block, which propagates the value of its input* on *to its output* powered *with a delay* Δ*. We therefore obtain* $t_{pow} - t_{on} = \Delta$*, where* t_{pow} *and* t_{on} *are, respectively, the times at which* powered *and* on *are asserted, and* Δ *is selected to accommodate the delay of the physical platform. Then, the designer implements the required functionality by allocating the* Controller *to its higher priority task, while guaranteeing a worst case switch-on time* $t_{on}^{max} = t_d - \Delta$ *to meet the deadline on the* powered *signal.*

While the functional platform described above is very convenient to explore different control strategies, it is not sufficient to determine the correctness of the final design. In fact, the satisfaction of the timing viewpoints heavily relies on the assumptions on the delay of the physical system, which can only be discharged by the implementation platform. The architecture of the implementation platform is shown at the bottom of Figure 3.4. The functionality of the Controller *is mapped to a hardware execution platform* HW*, while the* EPS *is modelled by a cascade of a first order filter with time constant* τ*, represented in the figure as an electrical network, and an ideal* Comparator *block, with reference voltage* v_{ref}*. If the filter output voltage* v *is larger than* v_{ref}*, the* Comparator *asserts its output* powered*. The reference* v_{ref} *corresponds to* 90% *of the final value* v_f *reached by* v *at steady state.*

To show that the implementation platform refines the specification platform, hence satisfies the system requirements, we can formalize the interaction between the two levels in terms of the composition $\mathcal{C}^t \otimes \mathcal{M}^t$ *between two timing contracts:*

- $\mathcal{C}^t = (\{\delta_{on}, t_{on}, t_{pow}\}, \delta_{on} \leq \Delta, (t_{on} \leq t_d - \Delta) \rightarrow (t_{pow} \leq t_d))$ *specifies the timing behavior of the specification platform, by emphasizing its* vertical assumptions *on the implementation platform. If the implementation platform provides a delay* δ_{on} *less than or equal to* Δ *when on is asserted, then the application guarantees to satisfy the requirement on the* powered *signal, if* on *is asserted by at least an interval* Δ *before the deadline* t_d*.*

- $\mathcal{M}^t = (\{\delta_{on}, t_{on}, t_{pow}\}, \mathtt{T}, (\delta_{on} = \tau \ln 10) \wedge (t_{pow} = t_{on} + \delta_{on}))$ *exposes the timing behavior of the implementation platform, as derived from the step response of a first-order filter.* $\mathcal{M}^t$ *states that, whenever* on *is asserted, the delay at which* v *reaches* 90% *of its steady-state level, hence* t_{pow} *is asserted, is* $\delta_{on} = \tau \ln 10$*.*

In this simple example, the assumptions and guarantees of both $\mathcal{C}^t$ *and* $\mathcal{M}^t$ *are assertions over variables denoting the time of occurrence of certain events or their separation.*

Resting on the above contracts, because the assumptions of $\mathcal{C}^t$ *trivially imply the ones of* $\mathcal{M}^t$*, we obtain* $\mathcal{C}^t \otimes \mathcal{M}^t \preceq \mathcal{C}^t$*. Moreover, for the contracts in this example, it is also*

possible to show that $\mathcal{M}^t \preceq \mathcal{C}^t \otimes \mathcal{M}^t$ *holds. Therefore, checking the correctness of the design finally requires checking that the vertical contract* $\mathcal{C}^t \otimes \mathcal{M}^t$ *is compatible. In this case, by applying* (3.16) *in Section 2.3.3, where* $\mathcal{M}^t$ *and* $\mathcal{C}^t$ *act, respectively, as* $\mathcal{C}_1$ *and* $\mathcal{C}_2$*, we conclude that it is enough to check the satisfiability of* $\forall \delta_{on} : \forall t_{pow} : (\delta_{on} = \tau \ln 10) \wedge (t_{pow} = t_{on} + \delta_{on}) \rightarrow (\delta_{on} \leq \Delta)$*, which provides*

$$\tau \ln 10 \leq \Delta. \tag{3.22}$$

This inequality can also be used at design time, as a practical guideline to dimension either the specification platform, by increasing its margin Δ*, or the implementation platform, by decreasing its time constant* τ*, to deploy a correct design.*

It is also possible to obtain the same result as in Example 20 by following an alternative formulation based on contract conjunction.

Example 21 (Vertical Contracts as Conjunction)**.** *We consider the same setup as in Example 20. However, we suppose that the designer chooses instead to describe the timing behaviors of the system in Figure 3.4 using a different contract pair:*

- $\tilde{\mathcal{C}}^t = (\{t_{on}, t_{pow}\}, t_{on} \leq (t_d - \Delta), t_{pow} \leq t_d)$*, the specification contract, is no longer bound to the implementation platform. It simply states that the requirement on* t_{pow} *is satisfied if* on *is asserted by at least an interval* Δ *before the deadline* t_d*.*
- $\tilde{\mathcal{M}}^t = (\{t_{on}, t_{pow}\}, \mathtt{T}, t_{pow} = t_{on} + \tau \ln 10)$*, the implementation contract, is also independent of the specification platform (except for being defined on the same variables), and exposes the timing behavior of the* powered *signal.* $\tilde{\mathcal{M}}^t$ *states that, whenever* on *is asserted,* powered *will be asserted with a delay* $\tau \ln 10$*, due to the physical system (a first-order filter).*

Then, to check the correctness of the refinement, a binding mechanism between the two contracts, each linked to its own platform, can now be provided by the conjunction of $\tilde{\mathcal{M}}^t$ *and* $\tilde{\mathcal{C}}^t$*.* $\tilde{\mathcal{C}}^t \wedge \tilde{\mathcal{M}}^t$ *ensures that both contracts are jointly satisfied, and refines* $\tilde{\mathcal{C}}^t$ *by construction. Therefore, all we need to check is that* $\tilde{\mathcal{M}}^t$ *does not create inconsistencies in* $\tilde{\mathcal{C}}^t \wedge \tilde{\mathcal{M}}^t$*, in the sense that* $(\forall t_{on} : \exists t_{pow} : G_{\tilde{\mathcal{M}}^t} \cap G_{\tilde{\mathcal{C}}^t})$ *is true*[6]*, where* $G_{\tilde{\mathcal{M}}^t}$ *and* $G_{\tilde{\mathcal{C}}^t}$ *are the guarantees of the two contracts in saturated form. In our case,*

$$\begin{aligned}
&\forall t_{on} : \exists t_{pow} : (t_{pow} = t_{on} + \tau \ln 10) \wedge ((t_{on} > t_d - \Delta) \vee (t_{pow} \leq t_d)) \\
&= \forall t_{on} : (\exists t_{pow} : t_{pow} = t_{on} + \tau \ln 10) \wedge (t_{on} > t_d - \Delta) \vee (t_{on} \leq t_d - \tau \ln 10) \\
&= \forall t_{on} : (t_{on} > t_d - \Delta) \vee (t_{on} \leq t_d - \tau \ln 10)
\end{aligned}$$

[6]We are actually interested in checking consistency $\forall t_{on} : t_{on} \leq (t_d - \Delta)$, which is the set of legal environments for $\tilde{\mathcal{C}}^t$. In fact, we want to show that, for each t_{on} satisfying the assumptions of the specification contract $\tilde{\mathcal{C}}^t$, there exists an implementable t_{pow}, according to the implementation contract $\tilde{\mathcal{M}}^t$, which also satisfies the deadline t_d, as required by $\tilde{\mathcal{C}}^t$. When $t_{on} > (t_d - \Delta)$, $\tilde{\mathcal{C}}^t \wedge \tilde{\mathcal{M}}^t$ is trivially consistent, since the guarantees of $\tilde{\mathcal{C}}^t$ are vacuously true.

leads to the condition $\tau \ln 10 \leq \Delta$*, which is the result found in* (3.22)*. Intuitively, this amounts to requiring that, if* t_{on} *and* t_{pow} *have to synchronize so that* $\tilde{\mathcal{M}}^t$ *refines* $\tilde{\mathcal{C}}^t$ *and the overall system satisfies the timing requirement on* t_{pow}*, then the delay implemented by the physical system in* $\tilde{\mathcal{M}}^t$ *must be smaller than or equal to the one defined by the application platform in* $\tilde{\mathcal{C}}^t$*.*

The approach illustrated above has been previously adopted in the design of analog and mixed-signal integrated circuits [155, 152], by leveraging effective approximations of implementation constraints to formulate vertical contracts representing different viewpoints (e.g., timing, energy, noise), and then checking their compatibility or consistency during design space exploration. More recently, a similar approach has also been advocated in the context of AUTOSAR [34]. Alternatively, when vertical assumptions and guarantees cannot be effectively expressed by compact models, compatibility and consistency of vertical contracts can be checked by co-simulation of the application and implementation platforms under a mapping mechanisms, such as the one in the METRONOMY framework [90], in which tuples of signals in the two platforms are synchronized. In the context of our example, this technique can be applied by unifying both occurrences and values of the *on* and *powered* signals, as shown in red in Figure 3.4, and then checking that the synchronized models satisfy the requirements.

We finally observe that the formal notion of vertical contracts we have presented is general, in that it encompasses other notions of contracts that were previously introduced in a control setting to capture the interactions between the controller and its execution platform [177, 73]. In this scenario, a controller takes as assumptions several aspects that include the timing behavior of the control tasks and of the communication between tasks, e.g., delay, jitter, as well as the accuracy and resolution of the computation (vertical assumptions in $\mathcal{C}$). On the other hand, the controller provides guarantees in terms of the amount of requested computation, activation times and data dependencies (vertical guarantees in $\mathcal{C}$). Vertical contracts can then be effectively used to formalize the agreement between control, software, and hardware engineers when specifying both system functionality and timing requirements. As a result, several design approaches and guidelines, which have been previously established in the literature in terms of "design contracts" [73], can be derived by formulating vertical contracts for both the software and control layers, and by enforcing their compatibility and consistency as illustrated by the example in Figure 3.4.

3.4.2.1 Top-Down and Bottom-Up Vertical Contracts in Platform-Based Design

In this section, we provide further insight into the use of vertical contracts in platform-based design, as reported by previous works in the literature [152, 177]. Our goal is to show how previous approaches can be smoothly framed within the contract framework proposed in this chapter, based on the notions of heterogeneous refinement and vertical contracts.

In PBD, horizontal contracts can be used to formalize the conditions for correctness of element integration at the same level of abstraction, while vertical contracts formalize the conditions for lower levels of abstraction to be consistent with the higher ones, and for abstractions of available components to be faithful representations of the actual parts. If vertical contracts are satisfied, the mapping mechanism of PBD can be used to produce design refinements that are correct by construction.

Informally, vertical contracts in PBD have often been decomposed into bottom-up and top-down contracts [152, 177]. When analyzing the behavior of complex cyber-physical systems, simplified macro-models are typically used to capture the relevant behaviors of the components at higher levels of abstraction. Therefore, guarantees should be provided on the accuracy of the macro-models with respect to models at lower levels of abstraction. These guarantees can be captured via *bottom-up vertical contracts*. On the other hand, in a top-down design approach, *top-down vertical contracts* can be used to encode top-level requirements that system architects introduce to craft the behavior of a chosen architecture according to the desired functionality. In the following, we show how both these concepts of bottom-up and top-down vertical contracts in the literature can be related to the notion of heterogeneous refinement and vertical contracts introduced in this chapter.

Specifically, when reasoning about a virtual system M at level $l+1$, a bottom-up vertical contract $\mathcal{C}_B^{l+1}$ can be used to capture what is expected to be offered by possible implementations of M at level l, so as to be able for M to perform its intended function at level $l+1$, as expressed by a top-down vertical contract $\mathcal{C}_T^{l+1}$. The correctness of the design using M at level $l+1$ will then depend on the existence of an implementation of M meeting this bottom-up vertical contract. Moreover, $\mathcal{C}_B^{l+1}$ adds to the horizontal contract $\mathcal{C}_H^{l+1}$, which is also attached to M, to capture the conditions imposed on its context at level $l+1$ for its correct integration. Such a structural breakdown into $\mathcal{C}_H^{l+1}$, $\mathcal{C}_B^{l+1}$ and $\mathcal{C}_T^{l+1}$ can be used to enforce orthogonalization of concerns, by separating function ($\mathcal{C}_T^{l+1}$) from communication ($\mathcal{C}_H^{l+1}$) and from implementation ($\mathcal{C}_B^{l+1}$). In this setup, $\mathcal{C}_T^{l+1}$ captures the top-level requirements on M, while $\mathcal{C}_H^{l+1}$ and $\mathcal{C}_B^{l+1}$ can be regarded as two different viewpoints, used to specify the conditions imposed, respectively, on the integration environment at level $l+1$ and the implementation platform at level l.

We now assume that the system M at level $l+1$ is to be implemented by an aggregation of subsystems $M_1, \ldots, M_n$ at level l, and show how the aforementioned structural decomposition can be leveraged to generate a hierarchy of verification (or synthesis) tasks during the design flow. When using budgeting in a *top-down approach*, the designer assigns responsibilities to the subsystems implementing M, by deriving top-down contracts $\mathcal{C}_{T1}^l, \ldots, \mathcal{C}_{Tn}^l$ for each of them. These contracts must jointly establish M's bottom-up vertical contract $\mathcal{C}_B^{l+1}$ by construction, and can be derived, for instance, by using synthesis methods. In this example, we assume that levels l and $l+1$ may use, in general, different behavior formalisms, which

are related by a mapping $\mathcal{M}$. We can then formalize the condition stated above as follows:

$$\left(\bigotimes_{i=1}^{n} \mathcal{C}_{Hi}^{l}\right) \bigwedge \left(\bigotimes_{i=1}^{n} \mathcal{C}_{Ti}^{l}\right) \preceq_{\mathcal{M}} \mathcal{C}_{B}^{l+1}, \tag{3.23}$$

where we highlight the fact that the execution of this cross-layer design step must assume that the integration of the different subsystems is successful, as prescribed by the horizontal contracts[7]. Alternatively, when using a *bottom-up approach*, the top-down vertical contracts of $M_1, \ldots, M_n$ at level l are given, and we need to establish that the bottom-up contract of M at level $l+1$ is satisfied by checking (3.23). Horizontal contracts can also be used this time as additional premise in the verification of refinement. The verification step in (3.23) can be performed in different ways. In particular, a convenient way could be to resort to a vertical contract, in the sense of Section 3.4.2, and prove that it is consistent. Given $\mathcal{C}_{B}^{l+1} = (V_{B}^{l+1}, A_{B}^{l+1}, G_{B}^{l+1})$, and $\mathcal{M}^{-1}(\mathcal{C}_{B}^{l+1}) := (R_{\mathcal{M}}^{-1}(V_{B}^{l+1}), \mathcal{M}^{-1}(A_{B}^{l+1}), \mathcal{M}^{-1}(G_{B}^{l+1}))$, such a vertical contract can be defined as follows:

$$\mathcal{M}^{-1}(\mathcal{C}_{B}^{l+1}) \wedge \left(\bigotimes_{i=1}^{n} \mathcal{C}_{Hi}^{l}\right) \bigwedge \left(\bigotimes_{i=1}^{n} \mathcal{C}_{Ti}^{l}\right),$$

which refines $\mathcal{C}_{B}^{l+1}$ by construction.

In both the top-down and bottom-up approach, the design finally proceeds with the additional verification steps required for each component M_i to demonstrate that, based on the expected capabilities of its realization, as expressed by its bottom-up vertical contract, the functionality of the component as expressed by its top-down vertical contract can be achieved, i.e.,

$$\forall i \in \{1, \ldots, n\} : \mathcal{C}_{Bi}^{l+1} \wedge \mathcal{C}_{Hi}^{l+1} \preceq \mathcal{C}_{Ti}^{l+1}. \tag{3.24}$$

Again, as shown in (3.24), these proofs can take the horizontal contracts of the components as additional supportive argument. Moreover, they can be performed by leveraging the vertical contracts $\mathcal{C}_{Ti}^{l+1} \wedge \mathcal{C}_{Bi}^{l+1} \wedge \mathcal{C}_{Hi}^{l+1}$, for each component M_i, and proving their consistency.

3.5 Conclusions

In this chapter, we developed the contract framework at the heart of our methodology for cyber-physical system design. Starting with the generic A/G contract theory presented in Section 2.3, we established a link between the theory of relational interfaces and the one of A/G contracts, shedding light on some of their key features for system specification, early detection of incompatibilities, and use of abstraction-refinement. Specifically, we proposed a

[7]We observe that the structural decomposition adopted in (3.23) is just an example. Another alternative could be to represent the left-hand side contract as $\bigotimes_{i=1}^{n}(\mathcal{C}_{Hi}^{l} \wedge \mathcal{C}_{Ti}^{l})$.

natural transformation from interfaces to LTL A/G contracts, and we found that it does not generally preserve serial composition and interface compatibility. To address this, we proposed a new projection operator on contracts that captures the distinct nature of inputs and outputs during hiding, thus enabling preservation of the semantics of interface composition and compatibility. We then formalized the concepts of heterogeneous refinement and vertical contract to deal with hierarchies of models characterized by both semantic and structural heterogeneity, i.e. using different formalisms and architectural decompositions. Such an augmented A/G contract framework, fully supporting multi-view and multi-layer design flows, is used to provide the formal foundations of the methodology developed in Chapter 4.

Chapter 4

Platform-Based System Design Methodology With Contracts

In this chapter we distill the main steps of our methodology, including architecture design and control design. For each step, we link to formalisms and tools that can be used to formalize requirements, build component and contract libraries, and map the specification into an implementation. We exemplify our integrated approach with CHASE*, a prototype environment designed to experiment with contract-based requirement formalization and manipulation.*

4.1 The Structure of the Methodology

We consider in this thesis a particular case of cyber-physical system (CPS) that incorporates most, if not all, of the features of general CPS, to help explain the methodology: a *control system*, composed of a physical *plant*, including sensors and actuators, and an embedded *controller*. The controller runs a *control algorithm* to restrict the behaviors of the plant so that all the remaining (closed-loop) behaviors satisfy a set of *system requirements*. Specifically, we consider *reactive controllers*, i.e. controllers that maintain an ongoing relation with their *environment* by appropriately reacting to it. Our goal is to design the *system architecture*, i.e. the interconnection among system components, and the control algorithm, to satisfy the set of high-level requirements. The CPS scenario of interest is pictorially represented in Figure 4.1.

As shown in Figure 4.2, the design methodology consists of two main steps, namely, system architecture design and control design. The *system architecture design* step instantiates system components and interconnections among them to generate an optimal architecture while guaranteeing the desired performance, safety, and reliability. Typically, this design step includes the definition of both the embedded system and the plant architectures. The *embedded system architecture* consists of software, hardware, and communication components, while the *plant architecture* depends on the physical system under control, and may consist of mechanical, electrical, hydraulic, or thermal components. Sensors and actuators reside at

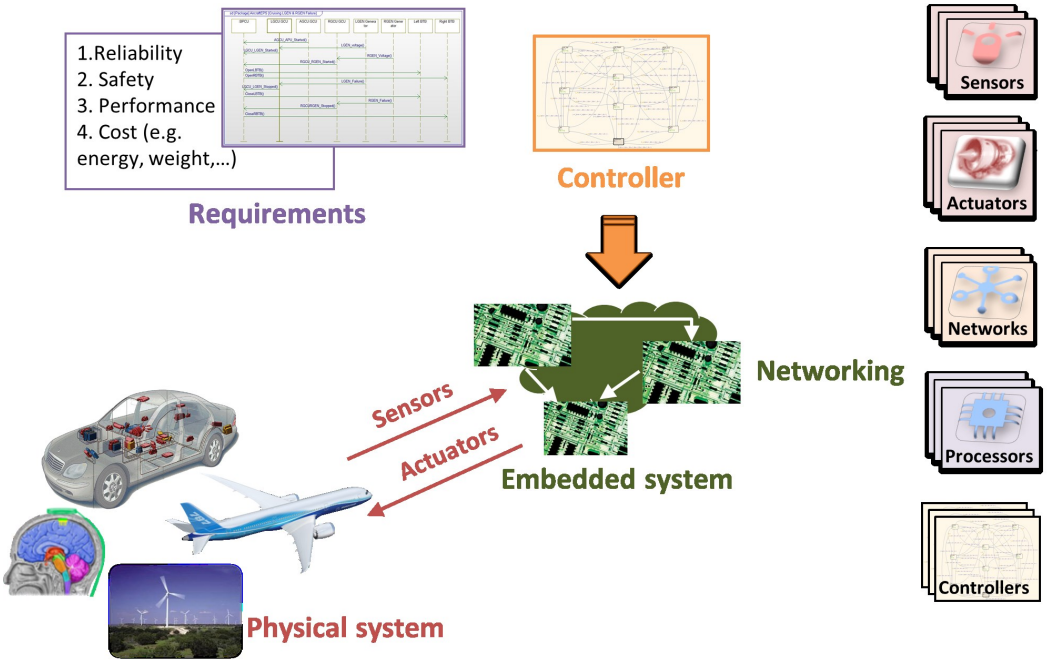


Figure 4.1: Pictorial representation of the class of cyber-physical systems considered in this dissertation.

the boundary between the embedded system and the plant [152]. Given an architecture, the *control design* step includes the exploration of the control algorithm and its deployment on the embedded platform.

The above two steps are however connected. The correctness of the controller needs to be enforced in conjunction with the assumptions on the plant. Similarly, performance and reliability of an architecture should be assessed for the plant in closed loop with the controller.

At the highest level of abstraction, the starting point is a set of requirements, predominantly written in text-based languages that are not suitable for mathematical analysis and verification. The result is a model of both the architecture and the control algorithms to be further refined in subsequent design stages. We place this process in the form of Platform-Based Design (PBD) and we use extensively the contract framework in Chapter 3 to verify the design and to build refinements that are correct by construction.

In PBD, the design is carried out as a sequence of refinement steps from a high-level specification to an implementation built out of a library of components at the lower level. Section 4.2 deals with requirement formalization and analysis for both architecture and control design. In Section 4.3, we provide a richer notion of platform component, and detail the development of the component (and contract) libraries for the methodology. Section 4.4

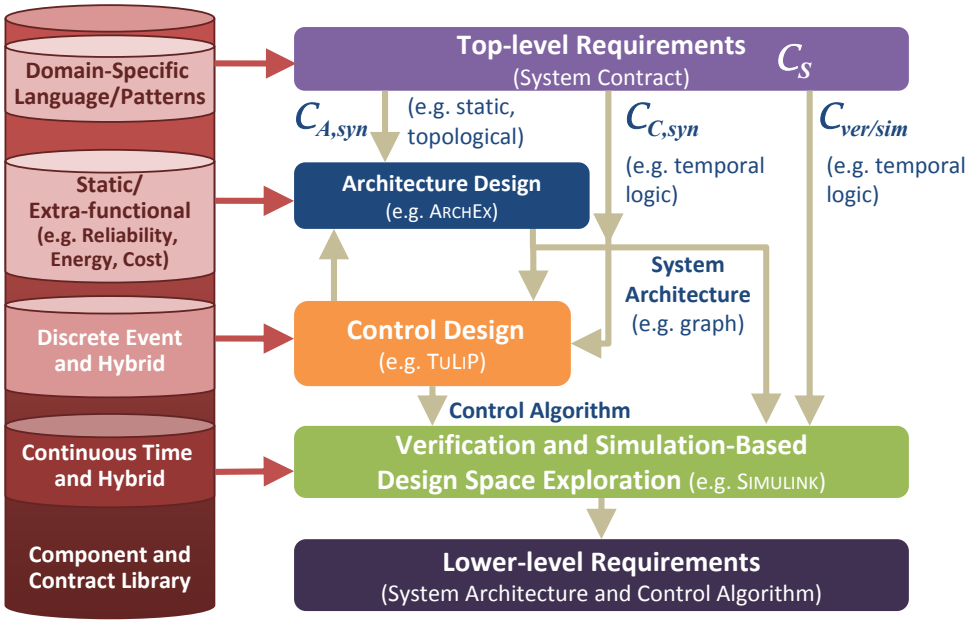


Figure 4.2: Structure of the proposed contract-based methodology for cyber-physical system design, from top-level requirements to the definition of system architecture and control algorithm (law).

delves into techniques for mapping requirements into an implementation. Finally, we present our experimental platform to benchmark our approach in Section 4.5 and draw some conclusions in Section 4.6.

4.2 Requirement Formalization and Validation

We use contracts to formalize top-level requirements, allocate them to lower-level components, and analyze them for early validation of design constraints. Requirement analysis can often be challenging, because of the lack of familiarity with formal languages among system engineers. Moreover, it is significantly different from traditional formal verification, where a system model is compared against a set of requirements. Since there is not yet a system at this stage, requirements themselves are the only entity under analysis. By formalizing requirements as contracts, it is instead possible to provide effective tests to check for requirement consistency, i.e. whether a set of contracts is realizable, or whether, in contrast, facets of these are inherently conflicting, and thus no implementation is feasible. Moreover, it is possible to exclude undesired behaviors, e.g. by adding more contracts, by strengthening assumptions, or by considering additional cases for guarantees. Since contracts are abstrac-

tions of components, their concrete representations are typically more compact than a fully specified design, as further illustrated in Chapter 6. The above tests can then be performed more efficiently than traditional verification tasks.

A framework for requirement engineering has been recently developed by leveraging modal interfaces, an automata-based formalism, as the underlying specification theory [34]. However, to retain a correspondence between informal requirements and formal statements, *declarative*, "property-based" approaches using some temporal logic are gaining increasing interest. They contrast *imperative*, "model-based" approaches, which tend to be impractical for high-level requirement validation. In fact, constructing a model to capture all the behaviors allowed by the requirements often entails considering all possible combinations of system variables. Moreover, these models are usually hard to parametrize, small changes in the requirements become soon hard to map into changes in the corresponding models.

As also mentioned in Section 2.3, we follow an approach based on the A/G contract framework as introduced in Chapter 3, which allows specifying different kinds of requirements using different formalisms, following both the declarative and imperative styles, to reflect the different viewpoints and domains in a heterogeneous system, as well as the different levels of abstraction in the design flow. As shown in Figure 4.2, to facilitate reasoning at the level of abstraction of requirement engineers, a viable strategy is to drive engineers towards capturing requirements in a structured form, using a set of predefined high-level *primitives*, or patterns, from which formal specifications can be automatically generated. This approach is similar to the one advocated in the STATEMATE verification environment [40], within the European projects SPEEDS and CESAR [65] (linked to automata-based formalisms), or to the higher-level *domain-specific language* (DSL) exemplified in Chapter 7.

From a set of high-level primitives, different kinds of contracts can be generated. When specifying the *system architecture*, steady-state (static) requirements, interconnection rules, component dimensions can be captured by static (stateless) contracts, expressed via *arithmetic constraints on Boolean and real variables* to model, respectively, discrete and continuous design choices. Then, compatibility, consistency and refinement checking translate into checking feasibility of conjunctions or disjunction of constraints, which can be solved via queries to Satisfiability Modulo Theory (SMT) solvers [28, 150] or mathematical optimization software, such as mixed integer-linear, mixed integer-semidefinite-positive, or mixed integer/non-linear program solvers.

When specifying the *control algorithm*, representing dynamic behaviors becomes the main concern; safety and real-time requirements can then be captured by contracts expressed using *temporal logic* constructs. In particular, linear temporal logic (LTL) [164] can be used to reason about the temporal behaviors of systems characterized by Boolean, discrete-time signals or sequences of events (discrete event abstraction in Figure 4.2). Signal temporal logic (STL) [136] can deal with dense-time real signals and continuous dynamical models (continuous abstraction in Figure 4.2). Sometimes, discrete and continuous dynamics are so tightly connected, that a discrete-event (DE) abstraction would result inaccurate, while a continuous abstraction would turn out to be inefficient, thus calling for a *hybrid system* abstraction, mixing discrete and continuous behaviors, such as Hybrid Linear Temporal

Logic with Regular Expressions (HRELTL) [61] and *hybrid automata* [11]. We reviewed the main formalisms and tools for the specification of dynamical systems in Section 2.4 and Section 2.6. These formalisms and tools can be used to implement the algebra of contracts and perform requirement analysis within our framework.

4.3 Platform Model-Library Development

In the bottom-up phase of the design process, a library of models of the components and the related contracts is developed for the plant and the embedded system. As shown in Figure 4.2, components and contracts are *hierarchically organized* to represent the system at different levels of abstraction, e.g. steady-state, discrete-event, and hybrid levels. Typically, at the highest levels of abstraction, a *signal-flow* approach is more appropriate to CPS modeling, as is the case in signal processing, feedback control based on sensor outputs and actuator inputs, and in systems composed of unilateral devices [202]. In these cases, relations between system variables are better viewed in terms of inputs and outputs, and interconnections in terms of output-to-input assignments. Inputs are used to capture the influence of the environment on the system, while outputs are used to capture the influence of the system on the environment. At the lowest levels of abstraction, *acausal* models, without a-priori distinction between inputs and outputs, may be more suitable to model the majority of physical (e.g. mechanical, electrical, hydraulic, or thermal) components, which are generally governed by laws that merely impose relations (rather than functions) among system variables, and where interconnections mean that variables are shared (rather than assigned) among subsystems. The A/G contract framework proposed in Chapter 3 can support both signal-flow and acausal models.

Reflecting the taxonomy of requirements, the model library is also *viewpoint and domain dependent*, following a similar approach as in the "rich component" libraries which were first proposed for automotive embedded systems [67]. At each level of abstraction, components are capable of exposing multiple, complementary viewpoints, associated with different design concerns and different formalisms (e.g. graphs, linear temporal logic, algebraic differential equations). Moreover, for platform components (and their contracts), models include extra-functional (performance) metrics, such as timing, energy and cost, in addition to the description of their behaviors, as further detailed in Section 4.3.1 below.

Components and contracts can then be expressed using the same formalisms introduced in Section 2.4 and Section 2.6, in the context of requirement analysis and system verification. As also mentioned in Section 1.1.3, a major challenge in this multi-view and hierarchical modeling scenario remains to maintain consistency among models and views, often developed using domain-specific languages and tools, as the library evolves. In this respect, the algebra of contracts can offer an effective way to incrementally check consistency or refinement among models. This information can then be stored in the library to speed up verification tasks at design time, as we will show in Chapter 6. Moreover, the mappings used in the definition of heterogeneous refinement and vertical contracts in Section 3.4 can also be used to establish

conditions for an abstract, approximate model, to be a sound representation of a concrete model, i.e. to define when a model still retains enough precision to address specific design concerns, in spite of the vagueness required to make it manageable by analysis tools (e.g., see the notions of top-down and bottom-up vertical contracts in Section 3.4.2.1).

We briefly reviewed the main languages and tools for system modeling and simulation, as well as a few attempts at their integration, in Section 2.5. In the following, since PBD is based on the composition of components while refining the design, we discuss a representation of a platform component that is richer than the one introduced in Section 2.3.1, in that it also includes extra-functional attributes in addition to the functional ones. To such a component we associate a set of properties that the component satisfies, and which are expressed with contracts. The contracts will be used to verify the correctness of the composition and of the refinements.

4.3.1 Platform Components

A *component* M can be seen as an abstraction representing an element of a design, characterized by the following *attributes*:

- a set of input U, output Y and internal X *variables* (including state variables); a set of configuration *parameters* K, and a set of input, output and bidirectional *ports* Λ. Components can be connected together by sharing certain ports under constraints on the values of certain variables. In what follows, to simplify, we use the same term variables to denote both component variables and ports;

- a set of *behaviors*, which can be implicitly represented by a dynamic *behavioral model* $\mathcal{F}(u, y, x, \kappa) = 0$, uniquely determining the values of the output ($y \in \mathcal{Y}$) and internal ($x \in \mathcal{X}$) variables given the values of the input variables ($u \in \mathcal{U}$) and configuration parameters ($\kappa \in \mathcal{K}$). Components can respond to every possible sequence of input variables, i.e. they are receptive to their input variables. Behaviors are generic and could be continuous functions that result from solving differential equations, or sequences of values or events recognized by an automata model. In the following, to simplify, we also use $[\![M]\!]$ to denote the set of behaviors of a component, as defined in Section 2.3.1;

- a set of *non-functional (extra-functional) models*, i.e. maps that allow computing non-functional attributes of a component, corresponding to particular valuations of its input variables and configuration parameters. Examples of non-functional maps include the *performance model* $\mathcal{P}(.) = 0$, computing a set of performance figures (e.g. bandwidth, latency) by solving a behavioral model, or the *reliability model* $\mathcal{R}(.) = 0$, providing the failure probability of a component.

Components can be hierarchically organized to represent the system at different levels of abstraction. A system can then be assembled by *parallel composition* and interconnection of components at level l, and represented as a new component at level $l + 1$. Moreover, by

exporting multiple models (both functional and extra-functional), which can be expressed via different formalisms and analyzed by different tools, a component exposes, at each level of abstraction, multiple, complementary *viewpoints*, associated with different design concerns (e.g. safety, performance, reliability).

A contract offers a *specification* for a component M. To simplify, we associate contracts with *non-parametric components*, i.e. components in which the configuration parameters K are fixed. Then, a *contract* $\mathcal{C}$ for a component M is a triple (V, A, G), where $V = U \cup Y \cup X$ is the set of component variables, and A and G are assertions, each representing a set of behaviors over V [33]. A represents the *assumptions* that M makes on its environment, and G represents the *guarantees* provided by M under the environment assumptions. All the definitions, operations and relations of the A/G contract framework in Section 2.3 and Chapter 3 will then hold. We provide examples of platform components and models below, and summarize the notation used for components and contracts in this thesis in Table 4.1 and Table 4.2.

Example 22 (Discrete Event and Hybrid Components for an Aircraft Power System)**.** *A power* generator *component at the discrete event (DE) level can be seen as a "source" component G_{DE}, whose output variable g communicates its healthy (1) or unhealthy (0) status. $\mathcal{F}_{G,DE}$ can be expressed as an automaton or an LTL formula describing the time behavior of g. $\mathcal{R}_G$ provides instead the failure probability p_G as a function of the flight duration.*

DE representations of generators, contactors, buses, rectifiers and loads can be composed and interconnected to build a DE representation P_{DE} of the power system plant*. At a lower abstraction level, a hybrid representation P_H of the plant can instead be assembled as shown in Figure 1.6 (g). $\mathcal{F}_{P,H}$ includes components described by algebraic differential equations. A trace obtained by simulating $\mathcal{F}_{P,H}$ includes three-phase voltage and current signals whose behaviors can be specified by STL formulas.*

The DE representation of the controller *M_{DE} includes as environment (input) variables the health status of generators, APUs, and rectifier units. Controlled (output) variables denote the contactors' status, and can each take values of open (0) or closed (1). Under the assumption of a synchronous system, the timing viewpoint of M_{DE} can be just captured by the clock period or reaction time T_r, which can be used as a configuration parameter to describe a family of possible controller implementations.*

4.4 Mapping Specifications to Implementations

In the absence of a unified framework for automated synthesis of CPS simultaneously subject to a heterogeneous set of requirements, we reason about different aspects or representations of the design by using specialized analysis and synthesis (mapping) frameworks that can operate with different formalisms. During design space exploration, both horizontal and vertical contracts can be used to define both the specification and the implementation platforms, thus playing an essential role in checking or enforcing that an aggregation of components is compatible, and that the implementation is a correct refinement of the specification.

Table 4.1: Notation: Platform components and contracts.

Symbol	Definition
M	Generic platform component
U	Input variable set
$\mathcal{U}$	Input variable domain
Y	Output variable set
$\mathcal{Y}$	Output variable domain
X	Internal (and state) variable set
$\mathcal{X}$	Internal variable domain
$S = U \cup Y \cup X$	Component (or system) variable set
$\mathcal{S}$, $dom(S)$	Component (or system) variable domain
K	Configuration parameter set
$\mathcal{K}$	Configuration parameter domain
Λ	Port set
$\mathcal{L}$	Port domain
$\mathcal{F}(u, y, x, \kappa) = 0$	Behavioral model
$[\![M]\!]$	Set of component behaviors
$\mathcal{P}(u, y, x, \kappa) = 0$	Performance model
$\mathcal{R}(u, y, x, \kappa) = 0$	Reliability model
$\mathcal{C} = (V, A, G)$	Contract
V	Contract variable set
$\mathcal{V}$	Contract variable domain
A	Contract assumptions (set of behaviors)
G	Contract guarantees (set of behaviors)
M	Contract implementation (component)
E	Contract legal environment (component)

At each abstraction level, *mapping* to a lower level can be performed by either leveraging a *synthesis* tool, or by solving an *optimization* problem that uses constraints from both the specification and the implementation layers to evaluate global tradeoffs among components. Accordingly, we denote as $\mathcal{C}_{syn}$ a contract that can be used as input of a specialized synthesis tool, and as $\mathcal{C}_{opt}$ a contract that serves as a conjunction of constraints in a more generic optimization problem. $\mathcal{C}_{opt}$ can be further characterized as $\mathcal{C}_{ver} \wedge \mathcal{C}_{sim}$, where $\mathcal{C}_{ver}$ denotes a contract whose satisfaction can be formally verified, e.g. using the tools introduced in Section 2.6, while $\mathcal{C}_{sim}$ refers to a contract that can only be checked by simulation. In the following, we first provide an overview of optimization-based mapping for design space exploration; then, we give examples of mapping techniques combining synthesis and optimization to perform the main design tasks in our methodology.

Table 4.2: Notation: Discrete Event (DE) and Hybrid abstractions.

	Discrete Event (LTL) Abstraction
Symbol	Definition
Σ	Generic system (component)
$[\![\Sigma]\!]$	System behavior set
$S = E \cup D$	System variable set
$\mathcal{S}$, $dom(S)$	System variable domain
s	System variable valuation (state)
$\sigma = s_0 s_1 s_2 \dots$	System behavior
E	Uncontrolled (environment) variable set
$\mathcal{E}$, $dom(E)$	Environment variable domain
e	Environment variable valuation
D	Controlled variable set
$\mathcal{D}$, $dom(D)$	Controlled variable domain
d	Controlled variable valuation
$\mathcal{C}_{LTL} = (S, \varphi_e, \varphi_e \rightarrow \varphi_s)$	LTL contract
	Hybrid Model (STL) Abstraction
Symbol	Definition
$\boldsymbol{u}(t)$ $(u(t))$	Input signal
$\boldsymbol{y}(t)$ $(y(t))$	Output signal
$\boldsymbol{x}(t)$ $(x(t))$	Internal (state) signal
$\boldsymbol{s} = (\boldsymbol{u}, \boldsymbol{y}, \boldsymbol{x})$ $(s(t) = \{u(t), y(t), x(t)\})$	System trace or behavior
$\boldsymbol{\kappa}$	Configuration parameter vector
$\mathcal{F}(\boldsymbol{u}, \boldsymbol{y}, \boldsymbol{x}, \boldsymbol{\kappa}) = 0$	Behavioral model
$\mathcal{C}_{STL} = (S, \varphi_e, \varphi_e \rightarrow \varphi_s)$	STL contract

4.4.1 Optimized Mapping and Design Space Exploration

In Section 2.7 we have discussed the challenges posed by the synthesis of a correct-by-construction system implementation, e.g. a controller, directly from a high-level formal specification (e.g. $\mathcal{C}_{C,syn}$ in Figure 4.2), and the tools developed over the years to address these challenges. Whenever correct-by-construction synthesis from requirements results into intractable problems, it is however possible to cast the design exploration problem, in its more general terms, as an optimization problem, where the system specifications are checked by a formal verification engine or by monitoring simulation traces.

For instance, let $\mathcal{C}_{sim} = (S, \phi_e, \phi_e \rightarrow \phi_s)$ be a contract that must be checked by simulation, where ϕ_e and ϕ_s are temporal logic formulas. Then, given an array of costs C, the mapping problem can be cast as a *multi-objective robust optimization* problem, to find a set of configuration parameter vectors κ^* that are Pareto optimal with respect to the objectives in C, while guaranteeing that the system satisfies ϕ_s for all possible traces s satisfying the

environment assumptions ϕ_e. More formally,

$$\min_{\kappa \in \mathcal{K}, \pi \in \Pi} \quad C(\kappa, \pi) \tag{4.1}$$
$$\text{s.t.} \begin{cases} \mathcal{F}(s, \kappa) = 0 \\ \\ s \models \phi_s(\pi) \qquad \forall s \text{ s.t. } s \models \phi_e(\pi) \end{cases}$$

where π is a set of formula parameters that can be used to capture degrees of freedom that are available in the system specifications, and whose final value can also be determined as a result of the optimization process. For a given parameter valuation κ', s' is shorthand notation for $s'(t) = (u'(t), y'(t), x'(t))$, the set of traces of input, output and internal signals (which are also represented as sets of traces over time $t \in \mathbb{R}_{\geq 0}$) that are obtained by simulating the behavioral model $\mathcal{F}(.)$, defined in Section 4.3.1. A multi-objective optimization algorithm with simulation in the loop can then be implemented to find the Pareto optimal solutions κ^*. While this may be expensive in general, it becomes the only affordable approach in many practical cases.

The mapping methodology above can also encompass contracts of the form $\mathcal{C}_{ver} = (S, \phi_e, \phi_e \rightarrow \phi_s)$ whose satisfaction can still be efficiently verified via formal methods, even if the synthesis problem is intractable. Moreover, it can be used to perform joint design exploration of the controller and its execution platform, while guaranteeing that their specifications, captured by vertical contracts, are consistent. Typically, the controller requirements are defined in terms of several aspects that are related to the execution platform, including the timing behavior of the control tasks and of the communication between tasks, their jitter, the accuracy and resolution of the computation, and, more generally, requirements on power and resource consumption. These requirements are taken as assumptions by the controller, which in turn provides guarantees in terms of the amount of requested computation, activation times and data dependencies. As mentioned in Section 3.4.2, the association of functionality to architectural services to evaluate the characteristics (such as latency, throughput, power, and energy) of a particular implementation by co-simulation of both a functional model and an architectural model of the system is supported by frameworks such as METRONOMY.

Finally, we observe that formal verification and synthesis algorithms usually operate on abstract and approximate representations of the design, which are only valid under certain assumptions. Some of these assumptions might not be discharged until high-fidelity models are considered. Therefore, a final set of simulations is often required to discharge any residual assumptions from previous design steps, and verify the overall design correctness, by monitoring the satisfaction of (some of) the contracts on high-fidelity models. Clearly enough, only by constructing appropriate abstractions at the early stages of the design process can we drastically reduce the number of final simulations and tests at the end.

4.4.2 Architecture Design

In the design of the system architecture, $\mathcal{C}_{A,syn}$ in Figure 4.2 includes the specification contract, e.g. expressed in terms of linear (or quadratic) arithmetic constraints on Boolean and real variables, as well as the steady-state models of the architecture, e.g. represented as constraints on a graph. Then, an implementation can be directly synthesized via optimized mapping, by solving a *mixed integer-linear (or quadratic) program* to minimize a cost function (e.g. component number, weight, cost, energy) while satisfying the constraints above. As shown in Chapter 5, the formulation above encompasses a variety of requirements, such as connectivity, safety, reliability, and energy balance. These requirements are mapped on a representation of the system architecture, e.g. in terms of a labelled graph, where nodes represent the (parametrized) components and edges represent their interconnections. Based on the specific shape of the resulting constraints, the mapping problem can then be solved by using either a "lazy" or an "eager" paradigm [92, 150].

The ArchEx framework introduced in Chapter 5 instantiates both of these paradigms to efficiently handle reliability requirements. Specifically, we implement two algorithms to decrease the complexity of exhaustively enumerating all failure cases on all possible graph configurations, namely, Integer-Linear Programming Modulo Reliability (ILP-MR) and Integer-Linear Programming with Approximate Reliability (ILP-AR). ILP-MR "lazily" combines an ILP solver with a background exact reliability analysis routine. The solver iteratively provides candidate configurations that are analyzed and accordingly modified to satisfy the reliability requirements. Although exact reliability analysis is an NP-hard problem, the idea is to perform it only when needed, i.e. a small number of times, and possibly on smaller graph instances. Conversely, ILP-AR "eagerly" generates a monolithic problem instance by leveraging approximate reliability computations that can still provide estimates to the correct order of magnitude, and with an explicit theoretical bound on the approximation error. The synthesized architecture can then serve as a specification for the control design step.

4.4.3 Control Design

For a given architecture, the controller requirements can be defined as a contract $\mathcal{C}_C$, where the assumptions A_C encode the allowable behaviors of the environment (including the physical plant) and the guarantees G_C encode the desired behaviors of the closed-loop system. Chapter 6 introduces two paradigms for systematic, contract-based design of control strategies, which merge optimized mapping methods with pre-existing control design and synthesis techniques.

The first paradigm, denoted as *Reactive Synthesis-Based Optimized Control Mapping (RS-OCM)*, enables the generation of hierarchical and distributed controller architectures by combining reactive synthesis from linear temporal logic contracts with simulation-based design space exploration, including monitoring of signal temporal logic contracts from simulation traces. In this paradigm, $\mathcal{C}_C$ can be expressed as the conjunction between an LTL contract $\mathcal{C}_{LTL}$ ($\mathcal{C}_{C,syn}$ in Figure 4.2) and an STL contract $\mathcal{C}_{STL}$ ($\mathcal{C}_{C,sim}$). The STL formulas

in $\mathcal{C}_{STL}$ can either be obtained by heterogeneous refinement of a subset of LTL formulas in $\mathcal{C}_{LTL}$ or generated anew to capture design aspects related to the plant and the hardware implementation of the control algorithm, which cannot be expressed using the Boolean, untimed, or DE abstractions offered by LTL. $\mathcal{C}_{LTL} \wedge \mathcal{C}_{STL}$ is then a *vertical contract* for the controller, since $\mathcal{C}_{LTL}$ and $\mathcal{C}_{STL}$ refer to two different controller representations, possibly involving different viewpoints (e.g. functional and timing).

To guarantee the consistency of $\mathcal{C}_{LTL} \wedge \mathcal{C}_{STL}$ and refine it towards an implementation, the controller design process consists of two steps:

1. *Reactive Synthesis.* As shown in Figure 4.2, $\mathcal{C}_{LTL}$ is first used together with DE models of the plant components (also described by LTL formulas) to synthesize a reactive control protocol in the form of one (or more) state machines using reactive synthesis techniques, as described in Section 2.7.1. A distributed (global) controller can also be built, in this phase, by composition from a library of pre-synthesized (local) controllers, using optimized mapping techniques. The resulting high-level controller will satisfy $\mathcal{C}_{LTL}$ by construction.

2. *Optimized Mapping.* The functional model of the synthesized controller is embedded into a high-fidelity hybrid model of the system, including an acausal representation of the plant. The entire system is simulated and the satisfaction of $\mathcal{C}_{STL}$ is assessed by monitoring simulation traces, while optimizing a set of system parameters and costs, as described in Section 4.4.1. The resulting optimal controller and plant configurations are returned as the final design, as shown in Figure 4.2.

We observe that the joint execution of the controller with the plant in the mapping step effectively implements the synchronization mechanism which is instrumental in: (i) checking the consistency of the vertical contract, (ii) discharging the timing assumptions made during the previous design steps, and (iii) ultimately verifying the satisfaction of both the functional and timing viewpoints. However, mapping via simulation may be expensive to perform for certain kinds of requirements; reactive synthesis is then key to make it affordable, by guaranteeing that several functional, safety and reliability requirements are already satisfied by construction.

The second paradigm, denoted as *Programming-Based Optimized Control Mapping (P-OCM)* uses, instead, a formalization of the design requirements and the plant model in terms of arithmetic constraints over real numbers, and formulates the control problem as an optimization problem that is solved within a receding horizon approach to determine a correct control policy that can also optimize some performance metrics. It is then possible to extend our contract-based approach to the design of model predictive control algorithms [84].

Specifically, in the P-OCM paradigm, we leverage a discrete time abstraction of the continuous behaviors of the system, and express $\mathcal{C}_C$ using either first order difference equations involving the component variables and parameters (time varying properties), or arithmetic constraints on real variables that must hold at each time step (time invariant properties). The

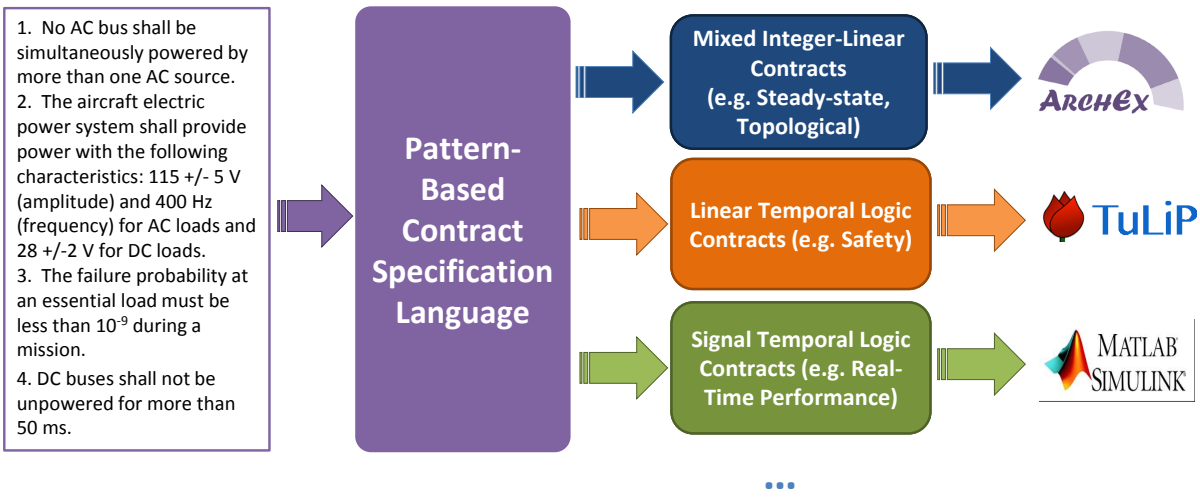


Figure 4.3: CHASE uses a contract specification language based on patterns to capture system requirements and facilitate their translation into formal specification languages for their analysis and validation.

algebra of contracts can then be implemented by simply combining constraints via conjunction or disjunction to express, respectively, intersections or unions of behaviors. Component models can also be developed by adopting the same discrete-time abstractions, i.e. by using difference equations for behavioral models, and polynomial constraints for performance and cost models. An optimal control problem can then be formulated aiming at minimizing the cost over a time horizon H, while satisfying the system dynamics and the contracts. Such a formulation, to be solved in a receding horizon fashion, is returned as the final design. Examples of this approach will be provided in Chapter 6 and Chapter 7, where we propose HOLMS, a hierarchical, optimal load management scheme for aircraft power systems based on an efficient mixed integer-linear program formulation.

4.5 CHASE: An Experimental Platform for Contract-Based Requirement Engineering

As shown in Figure 4.3, our methodology uses different formalisms to formalize and manipulate different kinds of requirements at different stages of the design process. As discussed before, we use automata or temporal logic constructs to express safety requirements for control design; we use arithmetic constraints on Boolean variables, to express structural requirements for architecture selection; we use linear or nonlinear real constraints on models governed by integro-differential equations, to express real-time requirements. Our goal is

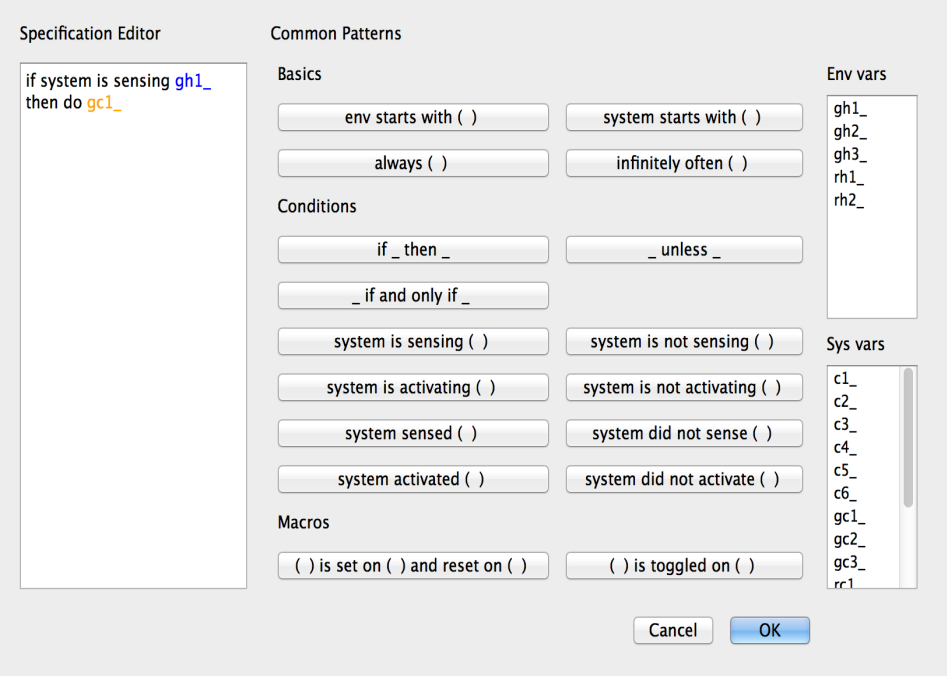


Figure 4.4: Screenshot showing the set of patterns for system specification in CHASE.

to orchestrate a set of analysis and synthesis tasks involving this heterogeneous set of formalisms by leveraging an appropriate combination of verification, synthesis, and simulation tools. Examples of tools may include ArchEx, used to reason about mixed integer-linear constraints, TuLiP, used for synthesis from LTL specifications, Simulink and Breach, used for monitoring STL specifications from simulation traces.

To experiment with contract-based requirement engineering and compositional design methodologies, as the one described in this chapter, we have developed the prototype framework *CHASE (Contract-Based Heterogeneous Analysis and System Exploration)*. CHASE leverages A/G contracts expressed using logic specification languages and a set of back-end synthesis and verification tools to analyze requirements and help reason about their correctness, completeness, and consistency.

As represented in Figure 4.3, CHASE adopts a structured front-end language for contract specification based on patterns to facilitate requirement capture and formalization. As described in Section 4.2, patterns can be regarded as templates, which can be easily used by system and requirement engineers, typically unfamiliar with formal specification languages, to encode their requirements. Translation into a set of mathematical constraints or logic formulas can then be done automatically. Figure 4.4 shows the set of patterns used to capture and translate requirements into LTL A/G contracts, which are supported by the

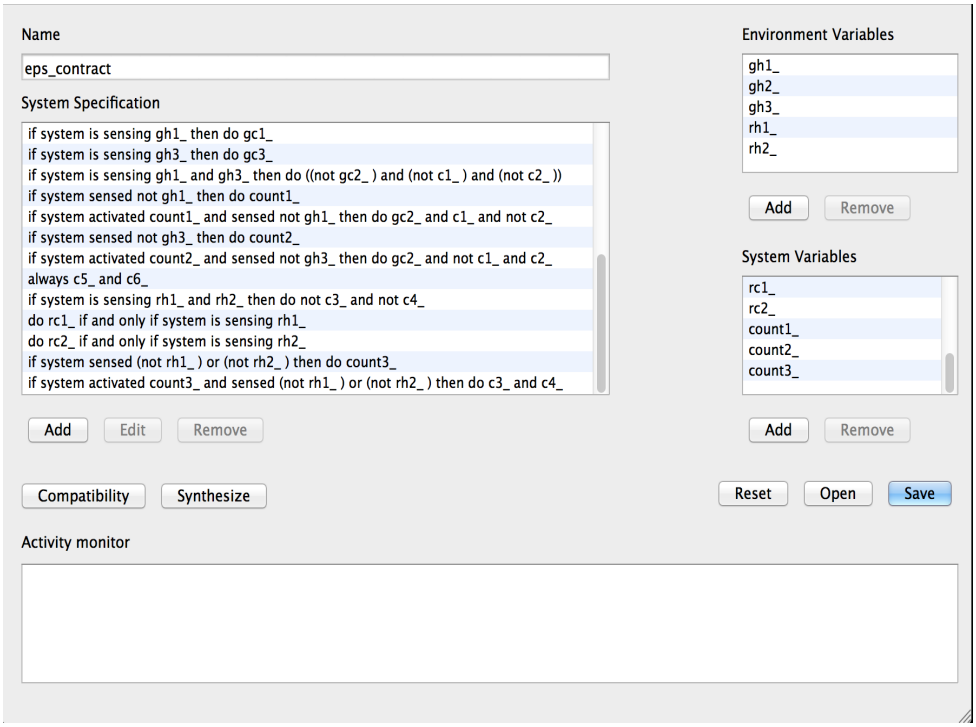


Figure 4.5: Given a set of environment variables, system variables and requirements, CHASE supports LTL satisfiability checks ("Compatibility" in the screenshot) and LTL realizability checks ("Synthesize" in the screenshot).

current PYTHON implementation[1]. CHASE provides a syntax-constrained editor to declare environment and system variables as well as the list of requirements using a combination of predefined primitives. The conjunction of requirements is internally converted into a contract which can be given as an input to different analysis and synthesis tools.

As shown in Figure 4.5, CHASE provides effective tests to check for requirement consistency, i.e. whether a set of contracts is realizable, or whether, in contrast, facets of these are inherently conflicting, and thus no implementation is feasible. To do so, it implements the algebra of LTL A/G contracts, as summarized in Section 2.4.1.4, by using the NUSMV model checker [59] to formulate and solve LTL satisfiability problems (e.g. used for refinement checking) and synthesis tools, such as TULIP [209] and ACACIA+ [44], to check for requirement realizability. It is then possible to interactively exclude undesired behaviors in a design, e.g. by adding more contracts, by strengthening assumptions, or by considering additional cases for guarantees. Finally, it is possible to incorporate higher-level domain-specific languages, as the one proposed in Chapter 7.

[1] A similar approach was proposed in the past for robotic motion planning applications (`http://ltlmop.github.io/`)

While still a prototype tool, CHASE offers already a meaningful example of how contracts can serve as a unifying formal framework for system design, by effectively coordinating different verification and synthesis tools to enable rigorous analysis of requirements including complex behaviors, and in a way that is easily accessible to system engineers.

4.6 Conclusions

We argued that system-level design of complex cyber-physical systems can be seen as a layered process with three main articulation points. After mapping a set of steady-state, structural, and reliability requirements into a high-level system architecture (architecture design step), a control design step is used to map functional, safety, and reliability requirements into a lower-level, discrete, dynamical representation of the system. Real-time performance requirements are finally mapped into higher-fidelity hybrid models by a simulation-based optimization step, which is also used to verify all the assumptions made in the earlier design stages. The above steps are all mediated by A/G contracts, and can be supported by an appropriate mix of formalisms and tools for requirement formalization, model library development, and optimized mapping. A proof-of-concept framework, CHASE, shows how contracts can effectively be used to coordinate different tools and enable rigorous analysis of complex behaviors in a way that is practically usable by system engineers. The remainder of the thesis delves into the single design steps and the results of their application to our case studies.

Chapter 5

Optimized Selection of Cyber-Physical System Architectures

This chapter deals with the design of safety-critical cyber-physical system architectures, by casting the design problem in terms of mapping of a higher-level "application" contract into a lower-level "implementation" contract. Mapping is formulated as a mixed integer-linear optimization problem on a reconfigurable graph which models the architecture, aiming at minimizing a cost function, while guaranteeing a desired bound on the reliability. To decrease the cost of generating symbolic probability constraints by exhaustive enumeration of failure cases on all possible graph configurations, we propose an efficient method to approximate the system reliability with an explicit bound on the estimation error. We then introduce the Integer-Linear Programming with Approximate Reliability (ILP-AR) and Integer-Linear Programming Modulo Reliability (ILP-MR) algorithms to solve the mapping problem, and demonstrate their effectiveness on the selection of aircraft electric power system architectures.

5.1 Introduction

Informally, a cyber-physical system (CPS) architecture can be seen as an interconnection of heterogeneous components assembled to perform a certain function. In a typical scenario, software components running on a hardware computing platform are connected in feedback with physical processes to form a large, distributed, control system subject to tight cost, safety, and reliability constraints. As mentioned before, the design of such a network is a challenging task, usually carried out based on pre-existing implementations and heuristic approaches, which hardly scale to the complexity of today's systems. System-level design exploration is indeed the domain of experienced architects, mostly relying on their accrued knowledge and a set of heuristic evaluations to take risky decisions. In fact, the result of *ad hoc* design practices is often at the origin of unsustainable delays, lengthy redesign cycles, and severe financial consequences.

In addition to the increasing complexity of these systems, a major obstacle to the devel-

opment of effective design tools for system-level architecture exploration is the heterogeneity of the design requirements, often expressed using different mathematical formalisms, and hard to be accounted for at once. It is, therefore, highly desirable to devise effective abstractions that enable co-design and optimization of CPS architectures under several, possible conflicting concerns, while guaranteeing design correctness and fault tolerance. In this chapter, we propose an optimization-based methodology for the selection of CPS architectures whose reliability is a function of the interconnection structure. Our goal is to minimize the overall system cost (e.g. number and weight of components) while guaranteeing that an upper bound on system failure probability is met. The contributions of this chapter can be summarized as follows:

- We provide a general graph representation of an architecture that allows an efficient casting of the design problem as a mixed integer-linear program (MILP), capable of modeling a variety of system requirements, such as connectivity, safety, reliability and energy balance.

- We propose an approximate, compositional, reliability computation method that alleviates the burden of exhaustively enumerating all failure cases on all possible graph configurations to provide a symbolic expression for the system failure probability as a function of the component failure probabilities and the interconnection structure. We denote this approximate computation method as *approximate reliability algebra*. While exhaustive enumeration has exponential complexity in the worst case, and results in the generation of nonlinear, non-convex, polynomial constraints, the approximate reliability algebra can instead generate *linear* reliability constraints, albeit with the introduction of auxiliary variables. However, the number of the additional variables (and associated auxiliary constraints) is polynomial, as opposed to exponential, in the size of the graph. Moreover, the algebra is particularly suitable for system-level design exploration, in that it generates estimations to the correct order of magnitude, and with an explicit, theoretical bound on the approximation error.

- We propose two algorithms to solve the optimal architecture selection problem: Integer-Linear Programming with Approximate Reliability (ILP-AR) and Integer-Linear Programming Modulo Reliability (ILP-MR). ILP-AR *eagerly* formulates a tractable, *monolithic* problem instance using the *approximate* algebra to generate the reliability constraints. Conversely, ILP-MR *lazily* combines an integer linear programming (ILP) solver (without reliability constraints) with an *exact* reliability analysis routine. The solver *iteratively* provides candidate configurations that are analyzed and accordingly modified, only when needed, to satisfy the reliability requirements. The approximate algebra is used by ILP-MR to generate additional constraints that can prune a large number of "conflicting" architecture configurations out of the search space whenever a reliability constraint is violated, thus dramatically decreasing the number of iterations and calls to the exact analysis routine.

- We implement the algorithms above in a prototype framework for CPS architectural exploration, ArchEx, and compare their performance on the design of architectures for aircraft electric power distribution.

The rest of the chapter is organized as follows. After a brief overview of related work in Section 5.2, we present the problem formulation in Section 5.3 and the approximate reliability computation method in Section 5.4. Section 5.5 and Section 5.6 introduce, respectively, the ILP-AR and ILP-MR algorithms, while Section 5.7 reports on their application to the power system case study. Lastly, conclusions are drawn in Section 5.8.

5.2 Related Work

In spite of the proliferation of techniques and tools for system reliability assessment, interoperability with automatic design exploration and optimization frameworks is still an open problem. In addition to the complexity of exact network reliability analysis, which is an NP-hard problem [130], techniques such as Fault Tree Analysis (FTA) or Reliability Block Diagrams (RBD) often rely on a set of system abstractions, which are hard to incorporate into system design flows [108]. For instance, in FTA, causal chains leading to some failure are depicted as a tree, inherently describing a hierarchical breakdown. However, in FTA, decomposition into modules mostly relates to the hierarchy of failure influences rather than to the actual system architecture. Therefore, the integration of fault trees with other system design models, or the automatic generation of fault trees from design artifacts, is not directly possible.

Differently than previous work, mostly focused on efficient methods for reliability analysis, the main focus of this chapter is on exploring approximate reliability computation techniques for architecture *synthesis*. For this purpose, we propose to evaluate reliability directly from the system structure, by associating a compact, albeit approximate, reliability model to each system component and interconnection, as also proposed by Kaiser et al. [108]. This *compositional* approach results into a practical abstraction to *concurrently* optimize for reliability and cost and guide design decision in a quantitative way at the system level. The ILP-AR algorithm aims to efficiently solve a *single optimization* problem, albeit of a large size, to provide an approximate solution to the architecture selection problem, without expensive calls to an exact reliability analysis function. On the other hand, instead of formulating a large, "flat" optimization problem, the ILP-MR algorithm avoids the expensive generation and manipulation of symbolic reliability constraints in the first place, via an iterative approach inspired by the lazy *ILP Modulo Theory* [92] or Satisfiability Modulo Theory (SMT) [28, 150] paradigms.

Helle et al. [93] have also proposed an approximate method for reliability calculations. However, our algebra is *richer*, since it accounts for the number of redundant paths implementing a certain function as well as the number of components of the same *type* that are actually used in these paths, as defined in Section 5.3. As a result, we can relax the

simplifying assumption made by Helle et al., that any used component is either maximally redundant (i.e., it participates in just one path) or essential (i.e., it participates in all paths).

5.3 Problem Formulation

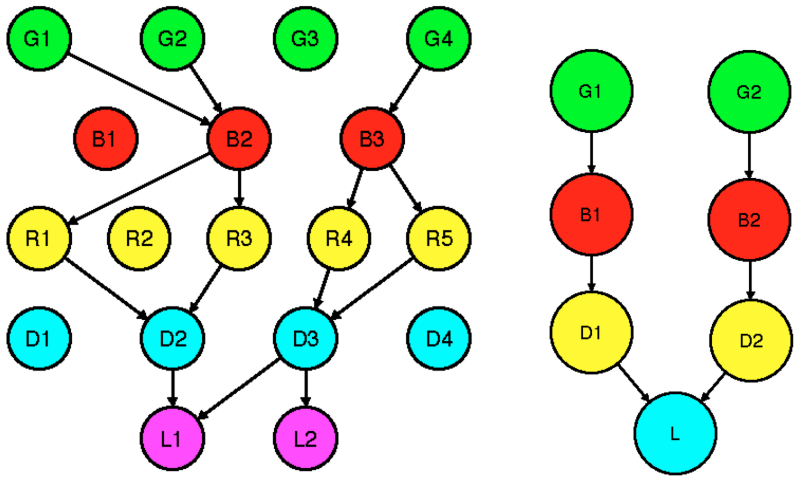
Consistently with the methodology described in Chapter 4, we assume that a design is assembled out of a *library* (collection) $\mathcal{L}$ of *components* and *contracts*. As discussed in Section 4.3.1, each component is associated with a set of *attributes*, which are used to capture both its functional and extra-functional properties, such as energy, performance, and cost. Components can be connected via *terminals* (also denoted as *ports* in Section 4.3.1) and terminal *variables*. At this level of abstraction, terminals are *logical* in nature. *Input* terminals are used to receive a signal or the value of a terminal variable; *output* terminals are used to send a signal or assign a value of a terminal variable. Contracts define, for each component and its environment, how terminals are connected and terminal variables are assigned. For the purpose of the architecture selection framework in this chapter, we can neglect the component dynamics, and express their behaviors in terms of stateless input-output relations. The same formalism is then used to define contract assumptions and guarantees. Specifically, each component in $\mathcal{L}$ is parametrized by a set of terminal variables $W = U \cup Y$, and a *type*, defining its functionality (role or task) in a system, as detailed below. Finally, each component is characterized by a cost c (cost model), and a failure probability p (reliability model). We can then formalize the notion of architecture as follows.

Definition 5.3.1 (Architecture)**.** *A system* architecture *is a directed graph $\mathcal{G} = (V, E)$, where V is a set of nodes and E is a set of edges. Both nodes and edges represent system components. Edge $e_{ij} \in E$ denotes the interconnection from v_i to v_j ($i, j \in \{1, \ldots, |V|\}$, $|V|$ being the cardinality of V)*[1].

In practice, every node and edge in an abstract architecture can be mapped to a library element that implements it; for instance, edges can be conveniently associated with *switches* to denote interconnections that are selectively activated. Nodes and edges can then be labeled with the same attributes as the associated library elements. A *template* $\mathcal{T}$ is an architecture, in which the number and types of nodes are fixed, while the interconnection structure is variable and can be reconfigured. In a template, edges can be represented by a set of Boolean variables $E = \{e_{ij}\}$, each denoting the presence or absence of an interconnection. An assignment over E defines an architecture *topology*. We then use the edge variables E of a template to formulate a topology selection problem. An example of architecture template is shown in Figure 5.1 (a), where different colors denote different types of nodes. Unconnected nodes are template elements which are not used in the final topology, determined by a specific assignment over E.

An architecture is assembled to perform one or more functions. We formalize this notion with the concept of *functional link*, involving a set of paths from source nodes to sink nodes,

[1] In the following, we also use e_{v_i,v_j} to denote e_{ij}.



(a) (b)

Figure 5.1: (a) Architecture template example: unconnected nodes represent components that are not used in the final topology; (b) Architecture analyzed in Example 24.

which are available to perform a certain function. We also assume that the link (edge) configuration can be modified by activating or deactivating paths from sources to sinks to react to component failures. The reliability of an architecture is then determined by its topological structure and the redundancy in the paths of the functional link. To define a functional link, we first introduce a partition on $\mathcal{T}$, with which we associate the notion of component type as follows.

Definition 5.3.2 (Graph Partition and Component Type)**.** *A* partition $\Pi = \{\Pi_1, \Pi_2, \ldots, \Pi_n\}$ *over the set of nodes* V *of* $\mathcal{T}$ *is a set of nonempty subsets of* V *such that* V *is a disjoint union of these subsets. We say that two nodes* a *and* b *have the same* type*, written* $a \sim b$*, when they belong to the same set in* Π*. If* a *is in* Π_i*, then we also say that its type is* i*.*

We also recall that a *walk* $\mu(v_a, v_b)$ of a graph $\mathcal{G}$ is a sequence of nodes $\{n_0, \ldots, n_k\}$ such that $n_0 = v_a$, $n_k = v_b$ and $e_{n_i n_{i+1}} \in E$ for each $i \in \{0, \ldots, k\}$. When all nodes in μ are distinct, we say that μ is a (simple) *path*, and write $|\mu|$ to denote the length of μ. Let Π_1 and Π_n be the subsets of V including, respectively, all sources and sinks. Then, a *functional link* F_i is the set of paths from any source in Π_1 to a sink $v_i \in \Pi_n$ that are used to perform an essential system function, on which a reliability requirement is given; in practice, such a function may consist in transferring data or energy from a source to the sink through a sequence of input-output links.

Example 23 (Functional Link)**.** *In Figure 5.1 (a) nodes of different colors represent different types. Source nodes at the top are in green, while magenta is used for sinks (at the bottom). The set of paths from source nodes to node* $L1$ *is a functional link, given by* {{G1, B2, R1,

D2, L1}, {G1, B2, R3, D2, L1}, {G2, B2, R1, D2, L1}, {G2, B2, R3, D2, L1}, {G4, B3, R4, D3, L1}, {G4, B3, R5, D3, L1}}.

Based on the definitions above, we cast the architecture selection problem as an optimization problem. Given a library $\mathcal{L}$ and a template $\mathcal{T}$, our goal is to derive a configuration that satisfies a set of interconnection and reliability requirements, while minimizing the cost and the complexity (number of components) of the overall network. The set of Boolean variables E will then include our decision variables. Based on the final assignment over E, some of the edges and nodes in $\mathcal{T}$ will be selected to generate an optimal architecture; unnecessary nodes and edges will instead be pruned away to minimize the overall cost. In our contract-based framework, the architecture is specified as an aggregation of contracts from the library $\mathcal{L}$, encoding some of the interconnection requirements, and which we denote as *implementation contract* $\mathcal{C}_\mathcal{T}$. The top-level requirements (both interconnection and reliability) are specified by a system-level *application contract* $\mathcal{C}_\mathcal{A}$. The refinement (mapping) between $\mathcal{C}_\mathcal{A}$ and $\mathcal{C}_\mathcal{T}$ is then modeled as the vertical contract $\mathcal{C}_\mathcal{A} \wedge \mathcal{C}_\mathcal{T}$, given by the conjunction of the implementation and application contracts. Therefore, we are interested in an optimal edge configuration e^* subject to the constraint that $\mathcal{C}_\mathcal{A} \wedge \mathcal{C}_\mathcal{T}$ is consistent, i.e. there exists an implementation satisfying both the guarantees of $\mathcal{C}_\mathcal{A} \wedge \mathcal{C}_\mathcal{T}$ in the context of their assumptions. In the following, we provide example formulations for the objective function and the Boolean arithmetic optimization constraints.

5.3.1 Objective Function

Let $\mathbf{e}$ be the adjacency matrix of $\mathcal{T} = (V, E)$, i.e. $e_{ij} = 1$ if there is one connection from v_i to v_j, and 0 otherwise. Then, the *objective function* can be expressed as the sum of the costs of all components (associated with both nodes and edges) used in the topology, i.e.

$$\sum_{i=1}^{|V|} \delta_i c_i + \sum_{i=1}^{|V|} \sum_{j=i+1}^{|V|} (e_{ij} \vee e_{ji}) \tilde{c}_{ij} \tag{5.1}$$

where c_i and $\tilde{c}_{ij}$ are the costs associated with node i and edge e_{ij}, respectively. δ_i is a binary variable equal to one if the node is instantiated in a configuration (topology) and zero otherwise. We express δ_i in terms of the edge variables as $\delta_i = \bigvee_{j=1}^{|V|}(e_{ij} \vee e_{ji})$, meaning that δ_i is one if there exists at least an edge (either ingoing or outgoing) between v_i and any other node in the graph (we assume $e_{ii} = 0$ for all i, and include it as an assumption in the contract of each node). Moreover, we use $(e_{ij} \vee e_{ji})$ when computing the cost associated with an edge, to avoid double counting the contribution of a component implementing a bidirectional interconnection. If two components with different costs are actually used to implement a bidirectional link, (5.1) can be modified accordingly.

5.3.2 Interconnection Constraints

A set of *interconnection* constraints originates from the component (horizontal) contracts in $\mathcal{L}$ and are used to enforce *legal connections* among components. For instance, the behavior of a component implementing node v_i may be defined by an *assertion*, such as $(y_i = \kappa_i)$, meaning that v_i assigns the value κ_i to its output y_i. Then, the contract for v_i may guarantee this behavior under the assumption that there is no self-loop, i.e. that its output is not connected to its input via an edge component. We then include the constraint $e_{ii} = 0$ in the optimization to require that the assumptions of v_i get discharged by the guarantees of its context (the other nodes and edges in the graph) for the composite contract $\mathcal{C}_\mathcal{T}$ to be compatible.

Another set of interconnection constraints derives instead from the application contract $\mathcal{C}_\mathcal{A}$, formalizing the requirements on the functional link and its set of paths to guarantee its *correct operation*. For example, let D, L, and B be subsets of V. Then, we can prescribe that there exists at least (most) one connection from a node in L and a node in D as follows:

$$\sum_{i=1}^{|D|} e_{l_j d_i} \geq (\leq)\ 1 \quad \forall\ j \in \mathbb{N} :\ 1 \leq j \leq |L|, \tag{5.2}$$

where $e_{l_j d_i}$ is the edge from node l_j to node d_i, and the inequality turns into an equality when one and only one connection is admitted. Moreover, we can state that if there exists an interconnection from any node in L to a node d_j in D, then d_j must be connected to at least one node in B, using a constraint of the form:

$$\bigvee_{1 \leq i \leq |L|} e_{l_i d_j} \leq \bigvee_{1 \leq k \leq |B|} e_{d_j b_k} \quad \forall\ j \in \mathbb{N} :\ 1 \leq j \leq |D|. \tag{5.3}$$

A third category of interconnection constraints can be used to enforce conservation laws or *balance equations* in physical systems, e.g. by requiring that the maximum power provided by a source in $\mathcal{T}$ is greater than or equal to the maximum power required by the connected sinks. Let d be a node in the graph, which is neither a source nor a sink. Let B be the set of direct predecessors of d, and L be the set of its direct successors. Then, a "local" balance equation at the terminals of d can be written as

$$\sum_{i=1}^{|B|} x_{b_i} e_{b_i d} \geq \sum_{j=1}^{|L|} y_{l_j} e_{d l_j}, \tag{5.4}$$

where x_{b_i} is the input value imposed by b_i and y_{l_j} is the output value assigned to l_j. Such a "local" assertion can be used, for instance, to express the guarantees of d. Alternatively, given a set of sources G and sinks Z, we can directly write a "global" balance equation for each source node as

$$y_{g_i} \geq \sum_{j=1}^{|Z|} x_{z_j} \eta_{g_i z_j} \quad \forall\ i \in \mathbb{N} :\ 1 \leq i \leq |G|, \tag{5.5}$$

where the path variable $\eta_{g_i z_j} = 1$ if there exists a path from source g_i to sink z_j, and 0 otherwise. Such path variables can then be related to the edge variables through a set of arithmetic constraints, as discussed in Section 5.5 and Lemma 5.5.1. Assertions as in (5.5) can be used to express the guarantees of a source node.

We have grouped together the set of constraints above, since they all generate linear arithmetic expressions in the decision variables, or include logical operations (conjunctions and disjunctions) that can be linearized with standard techniques [203]. The situation is different for *reliability* constraints.

5.3.3 Reliability Constraints

A typical reliability requirement prescribes that the failure probability of a sink, i.e. the probability that a sink gets disconnected from a source because of failures, should be less than a desired threshold. Therefore, to formulate a reliability constraint, as a part of $\mathcal{C}_{\mathcal{A}}$, we need to compute the probability of composite failure events in the system, starting from the failure probabilities of the components. Specifically, we denote as *system failure* R_i an event in which there is no possible connection between any of the available sources and a sink i, i.e. when the functional link F_i breaks, and as *reliability level* r_i the probability of R_i. Practically, the above notion of failure models the interruption of any information or energy transfer to an essential portion of the system. We assume that when a component fails, it cannot be recovered, and the adjacent links are no longer usable. Moreover, failures in different components are independent.

Let P_i be the event that component i fails (self-induced failure). Then, the event R_i of a system failure affecting component i can be recursively computed as follows

$$R_i = P_i \cup \bigcap_{1 \leq j \leq |V|, e_{ji} \neq 0} R_j, \tag{5.6}$$

where e_{ji} is j^{th}-row, i^{th}-column element of the adjacency matrix $\mathbf{e}$ of $\mathcal{T}$. In other words, component i fails when either a failure is generated in itself, or when failures are induced through its predecessors. A symbolic constraint for r_i can then be generated using (5.6) to enumerate all possible failure events while traversing $\mathcal{T}$ from node i to the sources, and then imposing $r_i \leq r^*$ to express the desired reliability requirement. However, such an exact computation, based on the enumeration of all possible component failure events, has exponential complexity on a fixed graph configuration [130]. The problem is further exacerbated when compiling a symbolic expression for a reconfigurable graph, since, in general, enumerating all possible configurations has also exponential complexity. Finally, the resulting reliability constraint would still include a nonlinear polynomial function of the decision variables (a sum of products of edge variables), and would either require a mixed integer-nonlinear programming (MINLP) solver, or a linearization step using a set of auxiliary variables and associated constraints that is linearly proportional to the number of products of Boolean terms in the original expression [203]. To overcome the above complexity issues, we propose

an approximate, more efficient, reliability computation method, and two algorithms for the solution of the optimal architecture selection problem based on it.

5.4 Approximate Reliability Computation

We propose an approximate algebra, where components contribute to the system failure probability based on their *degree of redundancy*, which is defined based on the notions of functional link and component type in Section 5.3. The system failure probability is then estimated as the sum of all the components' contributions. In this respect, the proposed reliability measure is additive and compositional.

We recall that any path in a functional link F_i consists of an interconnection of nodes, each having a role or performing a sub-task defined by its type. Nodes of the same type can be interchanged and introduce redundancy in the system architecture. We say that a type j, associated to a partition Π of $\mathcal{G}$, *jointly implements* a functional link F_i, written $\Pi_j \vdash F_i$, if all paths in the functional link F_i include at least one node in Π_j, i.e. $\Pi_j \vdash F_i$ iff $\forall \mu \in F_i : \mu \cap \Pi_j \neq \emptyset$. Trivially, we have $\Pi_1 \vdash F_i$ and $\Pi_n \vdash F_i$ for all i. Moreover, multiple nodes of the same type are allowed in a path as far as they are adjacent to each other. Given a path μ, possibly including multiple instances of the same type, we denote as $\hat{\mu}$ the *reduced path* obtained from μ after replacing all the instances of the same type with a single node, still of the same type. Therefore, after reducing all paths, each path μ_k in a functional link F_i, with $k \in \{1, \ldots, |F_i|\}$, is a sequence of distinct nodes $\{n_{k,1}, \ldots, n_{k,m+1}\}$, each of different type, interconnected through edges. Moreover, $n_{k,1}$ is a source ($n_{k,1} \in \Pi_1$) for all k, $n_{k,m+1}$ is the same sink node ($n_{k,m+1} = v_i$) for all k, and $n_{k,j} \sim n_{l,j}$ for all $k, l \in \{1, \ldots, |F_i|\}$, with $k \neq l$, and for all $j \in \{2, \ldots, m\}$[2].

Let c_{ij} the number of nodes of type j used in at least one reduced path of F_i, i.e. $c_{ij} = |(\cup_{\mu \in F_i} \hat{\mu}) \cap \Pi_j|$. We say that type j is *maximally connected* in F_i if each node of type j in F_i is connected to all the nodes of both the preceding and subsequent types in F_i. Clearly, if $c_{ij} = 1$ then type j is maximally connected. Let $I_i = \{j | \Pi_j \vdash F_i\}$ the set of component types jointly implementing F_i and $H_i = \{c_{ij} | j \in I_i, c_{ij} > 1\}$ the set of node numbers for all types with redundancy. We can then define the *degree of redundancy* h_{ij} associated with type j and link F_i as $h_{ij} = c_{ij}$ if type j is maximally connected; otherwise, we set $h_{ij} = \min H_i$. In other words, the degree of redundancy of type j is the number of nodes of type j if type j is maximally connected; if type j is, instead, not maximally connected, its degree of redundancy is the minimum number of nodes over all types with redundancy. Finally, we approximate the failure probability r_i of a functional link F_i by

$$\tilde{r}_i = \sum_{j \in I_i} c_{ij} p_j^{h_{ij}} \tag{5.7}$$

[2]Since the approximate algebra is mostly meant to be used for system-level architecture exploration, relying on reduced paths to define the approximate computation method is not a restrictive assumption. If needed, any node in a reduced functional link can be locally "refined" in terms of series interconnections of nodes of the same types at a later stage of the design process, by following a hierarchical approach.

where $I_i = \{j|\Pi_j \vdash F_i\}$ is the set of all component types that jointly implement F_i, p_j is the probability of failure of any of the components of type j, and h_{ij} is the degree of redundancy associated with type j and link F_i.

Example 24. *We illustrate the application of* (5.7) *to the architecture* $\mathcal{E}$ *represented in Figure 5.1 (b). We consider a partition where* $\Pi_1 = \{G_1, G_2\}$, $\Pi_2 = \{B_1, B_2\}$, $\Pi_3 = \{D_1, D_2\}$ *and* $\Pi_4 = \{L\}$. *Sink* L *is connected to sources* G_1 *and* G_2 *via two (reduced) paths, each using components of all the four types listed above. No type is maximally connected except for the sink; moreover we obtain* $\min H = 2$. *Therefore, the approximate expression for the failure probability of* L *would be* $\tilde{r}_L = p_L + 2p_D^2 + 2p_B^2 + 2p_G^2$, *while exact calculations lead to*

$$r_L = p_L + (1 - p_L)\{p_D + (1 - p_D)[p_B + (1 - p_B)p_G]\}^2 = p_L + (p_D + p_B + p_G)^2 + \text{h.o.t.},$$

where h.o.t. *(higher order terms) indicates higher order powers of the component probabilities. When all components are assumed to fail with the same probability* $p \ll 1$, *we obtain* $\tilde{r}_L = p + 6p^2$ *and* $r_L = p + 9p^2 + O(p^3)$.

The estimation in Example 24 has the same order of the exact calculation, and the error becomes negligible for small p. In general, (5.7) can provide "optimistic" estimations, but the bound to such optimism can be explicitly estimated for a given functional link. It is then possible to state the following theorem, providing a bound on the error of the approximate reliability algebra.

Theorem 5.4.1. *Given a graph* $\mathcal{G}$, *a partition* Π, *and a functional link* F, *let* $\tilde{r}$ *and* r *be, respectively, the approximate and exact failure probability for* F. *Let* $I = \{j|\Pi_j \vdash F\}$ *the set of component types jointly implementing* F. *We assume that the degree of redundancy is higher than one for at least one type in* I, *i.e.* $H = \{h_i|i \in I, h_i > 1\} \neq \emptyset$. *Then, the following inequality holds:*

$$\frac{\tilde{r}}{r} \geq \frac{h}{m^{h-1}}, \tag{5.8}$$

where $h = \min H$ *and* $m = |I| - 1$.

Proof. Without loss of generality, let $I = \{1, 2, \ldots, m, m + 1\}$, where the ordering in I denotes the sequence of types that jointly implement F, from sources (type one) to the sink (type $m + 1$). Let c_i be the number of nodes of type $i \in I$ used in F; by definition of F, we have $c_{m+1} = 1$. Moreover, we assume that a failure in the sink node v_{m+1} of F can only be induced by the failure of all the paths connecting it to any of the source nodes. Otherwise, since the degree of redundancy of v_{m+1} is one, i.e. v_{m+1} has no redundancy, the probability of a self-induced failure p_{m+1} would dominate the overall reliability of F.

To prove (5.8), we look for specific configurations of functional links for which the ratio $\tilde{r}/r$ is minimum. In this respect, we can immediately observe that any functional link in which $h_j = 1$ for at least one type $j \in I$, with $j \neq (m + 1)$ can be disregarded. In fact, it is straightforward to show that $\tilde{r} \geq r$ in this case, hence the approximate computation is conservative, and does not achieve the minimum $\tilde{r}/r$. Intuitively, this results stems from the

fact that, if any component in F has no redundancy, then the probability of a self-induced failure would dominate the overall reliability of F. In this scenario, the approximate algebra provides a result which is very close, and possibly larger, than the exact failure probability of F, hence $\tilde{r}/r$ is close to (or slightly larger than) one, and does not achieve its minimum. We shall prove this result analytically below.

Similarly, we can conclude that any maximally redundant functional link F_{max} can be disregarded, since it does not achieve the minimum ratio $\tilde{r}/r$. To do so, let P_v and R_v denote, respectively, the events of a self-induced and system-induced failure of a node $v \in V$. We aim to find an expression for $R^{max}_{v_{m+1}}$ in a maximally redundant functional link F_{max} that satisfies all of our hypotheses. Since interconnections are only allowed between consecutive types in F_{max}, according to the ordering in I, a maximally redundant link is obtained if and only if every node of type $i \in I$ is connected to all the nodes of type $i-1$, $i \in \{2, \ldots, m+1\}$. In a maximally redundant functional link, every type is maximally connected. It is then possible to prove, e.g. by induction, that, if a maximally redundant functional link includes m types of nodes (other than the sink), the event of a system failure at v_{m+1} can be computed as follows

$$R^{max}_{v_{m+1}} = \bigcup_{i=1}^{m} \left(\bigcap_{j=1}^{c_i} P_{ij} \right), \tag{5.9}$$

where P_{ij} is the failure event of node j of type i. Intuitively, in a maximally connected link, a sink node fails when at least one of the types fails to perform its task. On the other hand, for a type to fail, all the nodes of that type must fail. For such a configuration, we obtain $h_i = c_i$ for all i, which leads to the following approximate computation and bound:

$$\tilde{r}^{max} = \sum_{i=1}^{m} c_i p_i^{c_i} \geq \sum_{i=1}^{m} p_i^{c_i} \geq r^{max}, \tag{5.10}$$

where p_i is the failure probability of the nodes of type i. The approximate algebra is then conservative for a maximally redundant link. The approximate algebra provides a contribution that is also "pessimistic" for any maximally interconnected type in F. To find the minimum bound on $\tilde{r}/r$, we are interested, instead, in the most "optimistic" scenario, which occurs when no type in F is maximally interconnected, hence redundant.

Let F be one of such links, in which no type is maximally redundant. We further observe that for the minimum to be achieved, $h_i = h$ must also hold for all i, i.e. all the types in F should have the same (minimum) redundancy. In fact, if we had $h_i > h$ for some i in F, $\tilde{r}$ would increase with respect to the case in which $h_i = h$, since type i contributes with a coefficient $c_i = h_i > h$ in our approximate algebra, while p_i is raised to the same exponent h. Conversely, r would decrease with respect to the case $h_i = h$, because of the additional redundancy introduced by a higher number of nodes h_i in the link. Therefore, the ratio $\tilde{r}/r$ achieves its minimum when the functional link has the same number of components $c_i = h_i = h$ for all types. Moreover, the paths in F should be independent, i.e. they should not share any node other than the sink, since any additional interconnection of a node of

any type with additional nodes of adjacent types would result in the same $\tilde{r}$, but smaller r values, which would again lead to an increase in $\tilde{r}/r$. We then conclude that the ratio $\tilde{r}/r$ achieves its minimum on a functional link with h independent, parallel paths from a source to the sink.

Let F_{min} be such a link. For F_{min}, by using (5.7), we obtain

$$\tilde{r} = \sum_{i=1}^{m} h p_i^h, \tag{5.11}$$

while the exact calculation produces

$$r = \mathbb{P}\left(\bigcap_{j=1}^{h}\left(\bigcup_{i=1}^{m} P_{ij}\right)\right) = \left(1 - \prod_{i=1}^{m}(1-p_i)\right)^h, \tag{5.12}$$

that is, the sink fails when each of the h independent paths fail; moreover, a path fails if any node in the path fails. We can now rewrite (5.12) as follows:

$$r = \left(1 - (1-\bar{p})^m\right)^h, \tag{5.13}$$

to emphasize the dependence of r on an "average" path failure probability $\bar{p}$, defined such that $1-\bar{p}$ is the geometric mean of the probabilities $1-p_i$, each associated with a node in the path, i.e.,

$$\bar{p} = 1 - \left(\prod_{i=1}^{m}(1-p_i)\right)^{\frac{1}{m}}. \tag{5.14}$$

We observe that $\bar{p} \to (\sum_{i=1}^{m} p_i)/m$, i.e. $\bar{p}$ converges to the arithmetic mean of the p_i, and $r \to m^h \bar{p}^h$ as p_i tend to zero, for all i, which corresponds to the results obtained by neglecting the higher-order powers (i.e., higher than h) of the node probabilities in (5.12). The exact failure probability r of F_{min} is constant if $\bar{p}$ is constant. In particular, r would stay the same if $p_i = \bar{p}$ for all i. On the other hand, we can prove that there exists p^* such that, for all $p_i \leq p^*$, $i \in \{1, \ldots, m\}$, the following holds:

$$\tilde{r} = \sum_{i=1}^{m} h p_i^h \geq m h \bar{p}^h, \tag{5.15}$$

i.e., $\tilde{r}$ is always larger than or equal to the value achieved when $p_i = \bar{p}$ for all i, and reaches exactly its minimum when $p_i = \bar{p}$ for all i. Then, if (5.15) holds, we can also conclude that the minimum of $\tilde{r}/r$ is achieved when $p_i = \bar{p}$ for all i.

To show (5.15), it is enough to prove that

$$\left(\frac{1}{m}\sum_{i=1}^{m} p_i^h\right)^{\frac{1}{h}} \geq \bar{p}, \tag{5.16}$$

for all $p_i \leq p^*$, $i \in \{1, \ldots, m\}$. Moreover, by the monotonicity of the generalized mean function, we only need to show that (5.16) holds for $h = 2$. To do so, it can be shown that the function

$$f(p_1, \ldots, p_m) = \sqrt{\frac{1}{m}\sum_{i=1}^{m} p_i^2} - \bar{p}$$

is always larger than or equal to zero for all $p_i \leq p^* = 1 - (1/\sqrt{m})^{(m/(m+1))3}$ and achieves its minimum when $p_i = \bar{p}$ for all i. We observe that the selected bound p^* on the p_i is not tight, and can be improved for larger values of h and m. Moreover, while p^* converges to 1 as $m \to \infty$, we always have $p^*(m) \geq p^*(2) = 1/2$. Therefore, (5.15) is guaranteed to hold for all $p_i \leq 0.5$ independently of h and m. Since $p_i \leq 0.5$ is certainly true for component failure probabilities, we are allowed to conclude that the minimum of $\tilde{r}/r$ is indeed achieved when $p_i = \bar{p}$ for all i.

We are now ready to compute such a minimum $\tilde{r}/r$ as

$$\frac{\tilde{r}}{r} = \frac{mh\bar{p}^h}{\left(1 - (1-\bar{p})^m\right)^h}. \tag{5.17}$$

Then, by recalling that $a^n - b^n = (a-b)(a^{n-1} + a^{n-2}b + \cdots + b^{n-1})$ for $a, b \in \mathbb{R}$, $n \in \mathbb{N}$, we can write

$$\left(1 - (1-\bar{p})^m\right)^h = \bar{p}^h \left(1 + (1-\bar{p}) + (1-\bar{p})^2 + \cdots + (1-\bar{p})^{m-1}\right)^h \leq \bar{p}^h m^h, \tag{5.18}$$

which finally leads to the following bound, independent of $\bar{p}$,

$$\frac{\tilde{r}}{r} \geq \frac{mh\bar{p}^h}{\bar{p}^h m^h} = \frac{h}{m^{h-1}}, \tag{5.19}$$

as we wanted to prove. □

In the proof of Theorem 5.4.1, we have neglected the case in which $h_i = 1$ for at least one type $i \in \{1, \ldots, m\}$ in F. It is easy to show that our algebra is conservative in such a case, i.e. $\tilde{r} \geq r$. In fact, let i^* a type without redundancy, i.e. $h_{i^*} = 1$; then, the only available component v_{i^*} for type i^* is maximally connected, and can be considered in series with the two subgraphs F_{i^*-1} and F_{i^*+1} of the functional link F, including, respectively, all the paths from a source node to v_{i^*}, and from v_{i^*} to the sink. Let r_{i^*-1} and r_{i^*+1} denote, respectively, the failure probabilities associated with the subgraphs F_{i^*-1} and F_{i^*+1}. We obtain:

$$\tilde{r} = p_{i^*} + r_{i^*-1} + r_{i^*+1} \geq p_{i^*} + (1-p_{i^*})[r_{i^*-1} + (1-r_{i^*-1})r_{i^*+1}] = r. \tag{5.20}$$

As apparent from (5.20), for non-redundant types, our algebra is (slightly) conservative, its conservatism only stemming from the fact that higher-order terms (powers larger than one

[3] $f(p_1, \ldots, p_m)$ achieves its maximum in points $(p_1, \ldots, p_m)$ in which $p_i = 1 - (1/\sqrt{m})^{(m/(m+1))}$ for one index $i \in \{1, \ldots, m\}$ and $p_j = 0$ for all $j \in \{1, \ldots, m\}$ such that $j \neq i$.

in p_{i^*}) are disregarded. Our computation is, therefore, very accurate for small values of the failure probabilities.

Finally, while our focus so far has been on functional links associated with a single sink node, we observe that our results can be smoothly extended to scenarios involving a combination of functional links, possibly including multiple sink nodes, by carrying out the same computations for each link.

5.5 Integer Linear Programming With Approximate Reliability

The approximate reliability algebra in Section 5.4 allows efficiently encoding a reliability requirement into a set of *linear arithmetic constraints* on Boolean variables, while guaranteeing that the order of magnitude of the reliability calculations is correct, and there exists an explicit theoretical bound on the estimation error. Intuitively, the approximate reliability algebra relies on the fact that components with the highest failure probability tend to dominate the overall failure probability. In this section, we introduce the ILP-AR algorithm, which replaces exact reliability computations with the approximate algebra. While the expression in (5.7) is nonlinear, the additional number of constraints and auxiliary variables needed to linearize it is polynomial in the size of the template $\mathcal{T}$. In the following, we discuss the main results on correctness and complexity of ILP-AR.

The overall ILP-AR approach is illustrated in Algorithm 1. ILP-AR receives as inputs the library of components $\mathcal{L}$, together with their attributes, the template $\mathcal{T}$, and the set of requirements, including interconnection and reliability requirements. For the sake of compactness, in Algorithm 1, we use a vector notation for the library attributes and the required reliability values: $\boldsymbol{c}$ is the components' cost vector, $\boldsymbol{w}$ is the components' parameter vector (e.g., including generator power ratings and load power consumptions), $\boldsymbol{p}$ is the vector of components' failure probabilities, and $\boldsymbol{r^*}$ is a vector of required reliability (failure probability) values at critical nodes in the system. As mentioned in Section 4.2, in a design scenario, requirements can be inserted using patterns that can automatically generate interconnections and reliability constraints of the form discussed in Section 5.3.

To implement GenILP-AR, we use the approximate algebra to capture all the reliability requirements, which are then added to the other constraints as in Section 5.3. The resulting optimization problem is an ILP. While (5.7) is a nonlinear expression, a linear encoding of the same constraint can be obtained as follows

$$\sum_{l,k\in\{1,\ldots,k_{max}\},j\in\{1,\ldots,m\}} l \cdot x_{ijkl} \cdot p_j^k \leq r_i^*, \tag{5.21}$$

where r_i^* is the required failure probability, x_{ijkl} is an auxiliary binary variable equal to 1 if $j \in I_i$ (type j is used to implement F_i), $h_{ij} = k$ (the degree of redundancy of type j is k), and $c_{ij} = l$ (the number of nodes of type j in F_i is l), and 0 otherwise. k_{max} is the maximum possible value for h_{ij} (and c_{ij}) in the given template, i.e. $k_{max} = \max_{1\leq j\leq m} |\Pi_j|$.

Algorithm 1 *ILP With Approximate Reliability (ILP-AR)*

Input: Architecture template $\mathcal{T}$, vectors of component variables $\boldsymbol{w}$, costs $\boldsymbol{c}$ and failure probabilities $\boldsymbol{p}$ (from library $\mathcal{L}$), reliability requirements $\boldsymbol{r^*}$
Output: Adjacency matrix $\mathbf{e}^*$ of the final architecture $\mathcal{G}^*$

$(Cost, Cons) \leftarrow$ GenILP-AR($\mathcal{T}$, $\boldsymbol{w}$, $\boldsymbol{c}$, $\boldsymbol{p}$, $\boldsymbol{r^*}$)
$\mathbf{e}^* \leftarrow$ SolveILP($Cost$, $Cons$)
if $\mathbf{e}^* = [\,]$ **then return** `Infeasible`
return $\mathbf{e}^*$

Additional constraints are needed to express the auxiliary variables x_{ijkl} in terms of our decision variables. To show examples for some of these constraints, we assume that the reference template $\mathcal{T}$ only includes reduced paths. As mentioned earlier, this is not a restrictive assumption, since multiple instances of adjacent nodes of the same type can be added by refining $\mathcal{T}$ in a second step of the selection process. Next, we recall a lemma that allows reasoning about the existence of paths between any two nodes, which are possibly non adjacent, in a generic graph.

Lemma 5.5.1 (Walk Indicator Matrix)**.** *Let* $\mathbf{e}$ *be the adjacency matrix of a graph* $\mathcal{T}$*; let* $\mathbf{a} \odot \mathbf{b}$ *the* logical product *of two logical matrices* $\mathbf{a}$ *and* $\mathbf{b}$ *in* $\mathbb{B}^{m\times m}$*, defined as* $(\mathbf{a} \odot \mathbf{b})_{ij} = \bigvee_{k=1}^{m} a_{ik} \wedge b_{kj}$*; let* $\mathbf{e}^k = \underbrace{\mathbf{e} \odot \ldots \odot \mathbf{e}}_{k \text{ times}}$ *be the* k*-th* logical power *of* $\mathbf{e}$*. Then, the entry in row* i *and column* j*,* η_{nij}*, of the* walk indicator *matrix* $\boldsymbol{\eta}_n = \bigvee_{k=1}^{n} \mathbf{e}^k$ *is* 1 *if and only if there exists a directed walk of length less than or equal to* n *from vertex* v_i *to vertex* v_j*.*

Lemma 5.5.1 descends directly from the properties of the powers of the adjacency matrix of a graph, which can be extended to Boolean operators. By this lemma, to link the indicator variables x_{ijkl} to the decision variables, we need to add a set of constraints for each type j in $\{1, \ldots, m\}$. We first require that no more than one of the x_{ijkl} variables is asserted as follows

$$\sum_{l=0}^{k_{max}} \sum_{k=0}^{k_{max}} x_{ijkl} \leq 1. \tag{5.22}$$

We then define c_j as

$$c_j = \sum_{w \in \Pi_j} \left(\eta_{m_w, v_{m+1}} \wedge \left(\bigvee_{s \in \Pi_1} \eta_{m_s, w} \right) \right), \tag{5.23}$$

which computes the number of elements of type j used in F_i, and b_j as

$$b_j = \bigwedge_{y \in \Pi_j} \bigwedge_{w \in \Pi_{j-1}} \bigwedge_{z \in \Pi_{j+1}} \left((\delta_w \wedge \delta_y \rightarrow e_{w,y}) \wedge (\delta_y \wedge \delta_z \rightarrow e_{y,z}) \right), \tag{5.24}$$

where we define $\Pi_0 = \emptyset$ and $\Pi_{m+1} = \{v_{m+1}\}$, v_{m+1} being the sink node. b_j is a Boolean variable asserted if and only if type j is maximally connected. Constraint (5.23) counts the

number of components of type j which are connected by at least one path to v_{m+1} and to any source in Π_1. Constraint (5.24) uses the indicator variables δ introduced Section 5.3. δ_w is asserted if and only if node w is used in the link; its expression can be computed using the path indicator matrices of Lemma 5.5.1. Finally, we assume that a set of integer linear constraints are used to set the value for the variable c_{min}, defined as $c_{min} = \min\{c_j | c_j > 1, j = 1, \ldots, m\}$. We can then impose the following implications $\forall l \in \mathbb{N} : 0 \leq l \leq k_{max}$, $\forall k \in \mathbb{N} : 0 \leq k \leq k_{max}$,

$$\begin{aligned} (c_j = l) \wedge (b_j = 1) &\rightarrow x_{ijll} = 1 \\ (c_j = l) \wedge (b_j = 0) \wedge (c_{min} = k) &\rightarrow x_{ijkl} = 1. \end{aligned} \tag{5.25}$$

Constraints (5.25) guarantee that the indicator variable x_{ijkl} is set to 1 based on the definition of degree of redundancy associated to type j given in Section 5.4. All the implications and other logical operators in the above constraints can be converted into linear constraint using standard techniques [203]. Overall, the number of constraints (and auxiliary variables) generated by the computations in (5.21)-(5.25) is $O(|V|^3 m)$, where $m = |\Pi| - 1$, i.e. the number of types except for the sinks. This amounts to a polynomial complexity in the number of nodes and partitions in $\mathcal{T}$, which contrasts with the exponential complexity of exact computations. Finally, the following result holds for the ILP-AR approach.

Theorem 5.5.2 (Correctness of ILP-AR)**.** *For a given template $\mathcal{T}$, ILP-AR (Algorithm 1) is sound and complete within the error bounds provided by the approximate algebra.*

Proof. From the statement and the proof of Theorem 5.4.1, our approximate reliability measure can be either conservative or optimistic, based on the interconnection structure of the architecture, e.g. on whether a type of components is maximally interconnected or not, which is also dependent on the other optimization constraints. In general, let $l \leq 1$ and $u \geq 1$ be, respectively, the lower and upper bounds on the ratio $\tilde{r}/r$ between the approximate and the exact reliability measures. ILP-AR solves a "perturbed" optimization problem, in which reliability requirements of the form $r \leq r^*$ are replaced by perturbed requirements such as $r \leq \epsilon r^*$, with $\epsilon \in [1/u, 1/l]$. Therefore, ILP-AR can only be sound and complete on the perturbed problem. When ILP-AR provides an optimal topology, the actual reliability can be worst-case larger by $1/l$ with respect to the required one, where l is the lower bound (5.8). On the other hand, if ILP-AR returns `Infeasible` for a reliability requirement r^*, assuming that the interconnection constraints are feasible, we can only conclude that $r \leq r^*/u$ is infeasible. However, since ILP-AR attempts to determine, for each type of components, the degree of redundancy needed to meet the reliability requirement to the correct order of magnitude, in practical cases u is very close to one. □

5.6 Integer Linear Programming Modulo Reliability

The ILP Modulo Reliability (ILP-MR) algorithm avoids the expensive generation of symbolic reliability constraints by adapting the ILP Modulo Theory approach [92] to reliability

Algorithm 2 *ILP Modulo Reliability (ILP-MR)*

Input: Architecture template $\mathcal{T}$, vectors of component variables $\boldsymbol{w}$, costs $\boldsymbol{c}$ and failure probabilities $\boldsymbol{p}$ (from the library $\mathcal{L}$), reliability requirement r^*
Output: Adjacency matrix $\mathbf{e}^*$ of the final architecture $\mathcal{G}^*$

```
r ← 2r*
(Cost, Cons) ← GenILP(T, w, c)
while r > r* do                                   ▷ failure probability
    e* ← SolveILP(Cost, Cons)
    if e* = [ ] then return Infeasible
    r ← RelAnalysis(e*, p)
    if r > r* then
        Cons ← LearnCons(Cons, r, r*, e*)
    if Cons = [ ] then return Infeasible
return e*
```

computations, as summarized in Algorithm 2. ILP-MR receives as inputs the library of components $\mathcal{L}$, together with their attributes, the template $\mathcal{T}$, and the set of requirements, including interconnection and reliability requirements. As done in Algorithm 1, we also use a vector notation for the library attributes in Algorithm 2: $\boldsymbol{c}$ is the components' cost vector, $\boldsymbol{w}$ is the components' parameter vector (e.g., including generator power ratings and load power consumptions), and $\boldsymbol{p}$ is the vector of failure probabilities. Again, interconnections and reliability requirements can be inserted using patterns that automatically generate constraints as the ones shown in Section 5.3. To simplify, we assume that r is the worst case failure probability over a set of nodes of interest, for which the same reliability requirement r^* must be satisfied.

A smaller ILP problem, without reliability constraints, is solved in a loop with an exact reliability analysis routine. SolveILP generates minimum cost architectures for the given set of interconnection constraints. The RelAnalysis routine computes the probability of composite failure events at critical nodes, starting from the failure probabilities of the components, a problem known as K-terminal reliability problem in the literature [130]. To do so, we implement a modified depth-first search algorithm to traverse the graph $\mathcal{G}$ from the sink node i (root) to the source nodes (leaves), by applying a path enumeration method, and by turning event relations as in (5.6) into probability expressions. However, any other exact reliability analysis method for directed graphs can also be used [130]. Although the K-terminal reliability problem is NP-hard, the key idea is to solve it only when needed, i.e. a small number of times, and possibly on smaller graph instances.

At each iteration of ILP-MR, if the optimal architecture satisfies the reliability constraints, it is returned as the final solution. Otherwise, LearnCons estimates the number of redundant paths needed to achieve the desired reliability and suggests a set of strategies to implement the required paths by augmenting the original optimization problem with a set of interconnection constraints. This constraint learning function is, therefore, instrumental

Algorithm 3 LearnCons

Input: Current constraints $Cons$, reliability r, reliability requirement r^*, adjacency matrix $\mathbf{e}^*$

Output: Final constraints $Cons$

```
k ← EstPath(r, r*, e*)
NewCons ← [ ]
S ← GetSinks(e*)
(T_1, ..., T_n) ← GetTypes(e*)
for all v ∈ S do
    if k ≥ 1 then
        for all i ∈ (T_{n-1}, T_{n-2}, ..., T_1) do
            NewCons ← AddPath(v, i, k, NewCons, e*)
    else
        i ← FindMinRedType(v, e*)
        NewCons ← AddPath(v, i, 1, NewCons, e*)
    if NewCons = [ ] then return Infeasible
Cons ← Cons ∪ NewCons
return Cons
```

to efficiently converge towards a reliable architecture, while minimizing the number of calls to RelAnalysis. We provide details about this function in the following section.

5.6.1 Learning Constraints to Improve Reliability

When no reliability constraints are enforced in the ILP, the solver attempts to use the minimum number of components and interconnections to perform a specific function at minimum cost. Typically, such a "minimal" architecture has also minimal redundancy, hence minimal reliability. Based on this intuition, we develop strategies that increase the reliability of the solution, albeit at a higher cost, by enforcing a larger number of redundant components and interconnections. The overall routine is summarized in Algorithm 3.

Based on the current reliability level r, LearnCons estimates the number of additional redundant paths k required to satisfy the desired reliability r^* (function EstPath). As an example, under the assumption that all the paths in a functional link F are independent, then k can be computed as $k = \lfloor \log(r^*/r)/\log \rho \rfloor$, where ρ is the failure probability of a single path in F, and $\lfloor x \rfloor$ denotes the integer part of x. Since, in reality, the paths in F are not independent, this is a conservative estimation which avoids over-design. Then, if at least one additional path is required, for all the sinks and component types implementing F and used in the current architecture, AddPath generates new constraints to enforce that at least k additional components of each type have a path to the sink. These constraints do not necessarily translate into instantiating more nodes, as far as additional paths to the sink can

be obtained by just increasing the number of edges. If k additional paths cannot be obtained with the current template, AddPath attempts to enforce the maximum available number of paths. Conversely, if the estimated number of paths is zero, LearnCons attempts to still improve the overall reliability by enforcing one additional path between the sink and a component whose type has minimum redundancy in the current architecture, i.e. for which the total number of paths to the sink is minimum (as obtained from FindMinRedType). If no additional paths can be added between a sink and a component of any type, LearnCons terminates with `Infeasible`.

To enforce additional paths, AddPath uses the walk indicator matrix of $\mathcal{T}$, defined by Lemma 5.5.1. For example, we can require at least k additional connections of components of type T_i, belonging to the set Π_i of the partition Π of $\mathcal{T}$, to a sink v via at least one path of length $n - i + 1$ by enforcing

$$\sum_{w \in \Pi_i} \eta_{n-i+1 w,v} \geq k + \sum_{w \in \Pi_i} \eta^*_{n-i+1 w,v}, \tag{5.26}$$

where η_{n-i+1} and η^*_{n-i+1} are the walk indicator matrices, respectively, for $\mathcal{T}$ (decision variables) and the current architecture $\mathcal{G}^*$. The constraint (5.26) can be converted into an equivalent set of linear constraints in the elements of $\mathbf{e}$ (edge variables) by using standard linearization techniques. The following result summarizes the properties of the ILP-MR approach.

Theorem 5.6.1 (Correctness of ILP-MR)**.** *For a given template $\mathcal{T}$, if* SolveILP *is sound and complete on its problem instances, and* RelAnalysis *is exact, then ILP-MR (Algorithm 2) is sound and complete.*

Proof. Without loss of generality, we can focus on just one functional link F with a reliability requirement r^*. At each iteration k of the ILP-MR algorithm, based on Algorithms 2 and 3, a new architecture $\mathcal{G}_k$ is proposed by SolveILP, which is characterized by a failure probability r_k and a cost c_k. For example, $\mathcal{G}_1$ is the minimum-cost, minimally redundant architecture that is compatible with the original interconnection constraints. Since LearnCons enforces a minimum bound on the number of paths between the sink node and the set of nodes of one or more types, by encouraging the addition of new interconnections or components, as in (5.26), we conclude that c_k is a nondecreasing sequence, while r_k is a decreasing one.

We observe that, since the number of components in $\mathcal{T}$ is finite, the ILP-MR routine will terminate. Moreover, because RelAnalysis implements an exact reliability analysis method, if a final architecture is found, i.e. $r_k \leq r^*$ for some k, then it will satisfy all the requirements. ILP-MR is then sound.

On the other hand, ILP-MR can terminate with `Infeasible` when either LearnCons or SolveILP terminates with `Infeasible`. In the former case, because all the available paths will be eventually activated to increase the reliability, we infer that ILP-MR has exhausted all the paths, thus achieving the maximum redundancy allowed by the template. In the latter case, we infer that SolveILP fails to implement one (or more) of the paths that are

Table 5.1: Components and attributes used in the aircraft electric power system example.

Generators	g (kW)	Loads	l (kW)	Components	c
LG1	70	LL1	30	Generator	g/10
LG2	50	LL2	10	Bus	2000
RG1	80	RL1	10	Rectifier	2000
RG2	30	RL2	20	Contactor	1000
APU	100				

essential to increase reliability at iteration k, since the associate constraint is incompatible with any of the previous interconnection constraints. In both cases, we can conclude that, for the given template, there is no architecture which is able to satisfy all the constraints and ILP-MR is complete. □

5.7 Aircraft Power System Architecture Design

We apply our algorithms to the selection of optimal architectures for power generation and distribution in a passenger aircraft. A sample architecture, in the form of a single-line diagram, was introduced in Figure 1.3, together with the overall system description, and a qualitative discussion of the main design requirements in Section 1.2.3. We aim to generate an electrical power system (EPS) architecture that satisfies a set of connectivity, power flow, and reliability requirements while minimizing the total cost. We then model the EPS architecture as a directed graph, where each node represents a component (with the exception of contactors, which are associated with edges) and each edge represents an interconnection. An edge is directed from node v_i to node v_j if v_i receives power by (or through) v_j when traversing the graph from a critical load to a generator. We assume a template $\mathcal{T}$ consisting of the following component types: generators (LG/RG), AC buses (LB/RB), rectifiers (LR/RR), DC buses (LD/RD), loads (LL/RL), two on each side (right or left), and one APU. The platform library attributes include generator power ratings $\boldsymbol{g}$, load power requirements $\boldsymbol{l}$, component costs $\boldsymbol{c}$, and failure probabilities $\boldsymbol{p}$, as summarized in Table 5.1. We use a vector notation to denote the component attributes as done in Algorithm 1 and Algorithm 2. Moreover, in the examples of this chapter, we assume that only generators, buses, and rectifiers fail with a probability[4] of 2×10^{-4}.

[4]Experimental data on the failure rates of the physical components in an aircraft electric power system (e.g. contactors, generators, buses) have been collected over the years and made available in the literature. To relate failure rates with probabilities, we assume that the time at which a component can fail is a random variable with an exponential distribution, whose parameter λ is the failure rate [133]. Therefore, the probability that a failure is observed in a time interval T, e.g. given by the duration of a mission, can be computed as $P_{fail} = 1 - \mathrm{e}^{-\lambda T}$.

Table 5.2: Connectivity sub-matrices used in the aircraft electric power system example.

Variables	Interconnection	Dimension
$\mathbf{M}^{gb}$	Generator - AC Buses	$n_{gen} \times n_{acb}$
$\mathbf{M}^{bb}$	AC Buses - AC Buses	$n_{acb} \times n_{acb}$
$\mathbf{M}^{br}$	AC Buses - Rectifiers	$n_{acb} \times n_{rec}$
$\mathbf{M}^{rd}$	Rectifiers - DC Buses	$n_{rec} \times n_{dcb}$
$\mathbf{M}^{dd}$	DC Buses - DC Buses	$n_{dcb} \times n_{dcb}$
$\mathbf{M}^{dl}$	DC Buses - Loads	$n_{dcb} \times n_{load}$

5.7.1 Implementation and Application Contracts

In this section and in the rest of the thesis, we denote as $\mathcal{C}_T$ the application contract[5] for an aircraft electric power system, formalizing the load reliability requirements and power requirements (in nominal conditions), to emphasize that it relates to the system topology. On the other hand, the architecture contract $\mathcal{C}_\mathcal{T}$ formalizes the composition rules and interconnection requirements for an aggregation of contracts in $\mathcal{L}$, according to the interconnection structure E and the template $\mathcal{T}$, to be compatible. As discussed in Section 5.3 and Section 5.5, both the assumptions and the guarantees of $\mathcal{C}_T$ and $\mathcal{C}_\mathcal{T}$ can be concretely expressed using mixed integer-linear constraints in the decision variables and the parameters (attributes) associated with the graph elements (both nodes and edges). In what follows, we provide examples of assertions which instantiate, for the case of an aircraft EPS, the constraints presented in a generic form in Section 5.3 and Section 5.5. These assertions can be used to express either assumptions or guarantees in both the application and architecture contracts $\mathcal{C}_T$ and $\mathcal{C}_\mathcal{T}$.

To simplify our notation, we partition the adjacency matrix $\mathbf{e}$ of $\mathcal{T}$ into smaller blocks to represent interconnections between subsets of components, as summarized in Table 5.2. For instance, the interconnections between n_{gen} generators and n_{acb} AC buses can be represented by a $n_{gen} \times n_{acb}$ connectivity sub-matrix denoted as $\mathbf{M}^{gb}$. The *cost function* is the sum of the costs of all components (associated with the nodes) and contactors (associated with the edges) used in the electric power system architecture, as in (5.1). *Connectivity* properties can be expressed by using constraints as the ones in (5.2) and (5.3). For instance, we can prescribe that any rectifier must be directly connected to only one AC bus, and that all DC buses that are connected to a load or another DC bus must be connected to at least one rectifier to receive power from an AC bus. Using our notation, the former constraints can be written as follows

$$\sum_{i=1}^{n_{acb}} M^{br}_{i,j} = 1 \quad \forall\, j \in \mathbb{N} :\ 1 \leq j \leq n_{rec},$$

while the latter can be encoded by writing that the following constraints hold $\forall\, j \in \mathbb{N} : 1 \leq$

[5]Generically called $\mathcal{C}_\mathcal{A}$ in Section 5.3.

$j \leq n_{dcb}$,

$$\bigvee_{i=1}^{n_{rec}} M_{i,j}^{rd} \geq \bigvee_{i=1}^{n_{load}} M_{j,i}^{dl}, \qquad \bigvee_{i=1}^{n_{rec}} M_{i,j}^{rd} \geq \bigvee_{i=1}^{n_{dcb}} M_{j,i}^{dd}.$$

Similarly, we can enforce that all TRUs that are connected to a DC bus must be connected to at least one AC bus, i.e. $\forall\ j \in \mathbb{N} :\ 1 \leq j \leq n_{rec}$,

$$\bigvee_{i=1}^{n_{acb}} M_{i,j}^{br} \geq \bigvee_{i=1}^{n_{dcb}} M_{j,i}^{rd},$$

and all AC buses that are connected to a TRU or another AC bus must be connected to one generator, i.e. $\forall\ j \in \mathbb{N} :\ 1 \leq j \leq n_{acb}$,

$$\bigvee_{i=1}^{n_{gen}} M_{i,j}^{gb} \geq \bigvee_{i=1}^{n_{rec}} M_{j,i}^{br}, \qquad \bigvee_{i=1}^{n_{gen}} M_{i,j}^{gb} \geq \bigvee_{i=1}^{n_{acb}} M_{j,i}^{bb},$$

while we can use a constraint as the one in (5.2) to express that a rectifier cannot be directly connected to more than one DC bus and to more than one AC bus, i.e.,

$$\sum_{i=1}^{n_{dcb}} M_{j,i}^{rd} \leq 1, \quad \sum_{i=1}^{n_{acb}} M_{i,j}^{br} \leq 1, \quad \forall\ j \in \mathbb{N} :\ 1 \leq j \leq n_{rec}.$$

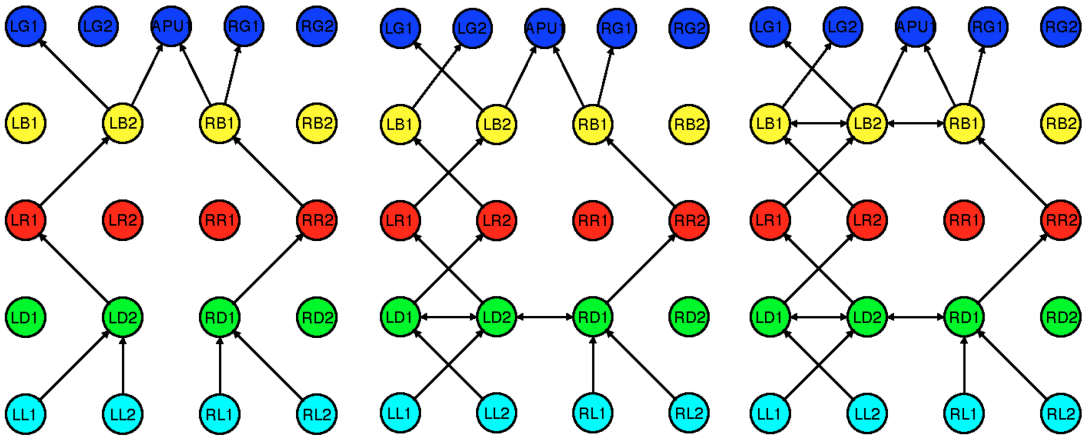
In Chapter 7 we will provide a few examples of patterns, i.e. high-level primitives that help formalize the power system topology requirements in terms of constraints as the ones above.

Power-flow constraints are used to enforce that the total power provided by the generators in each operating condition is greater than or equal to the total power required by the connected loads, by using expressions as in (5.4). For instance, in normal operating conditions, the power generated on each side should be greater than or equal to the total power required by the loads on that side. On the other hand, when only the APU is active, then it should be capable of powering at least the non-sheddable loads on both sides of the system.

Finally, a *reliability constraint* prescribes that the probability that a load gets unpowered because of failures should be less than a desired threshold. A functional link will then consist of the set of paths from any generator to the load. Moreover, since our template supports only reduced paths, we use an edge between two nodes of the same type as a shorthand notation to indicate two redundant components: if v_i and v_j, with $v_i \sim v_j$, are connected by an edge, then any direct predecessor of v_i is also a direct predecessor of v_j and *vice versa*.

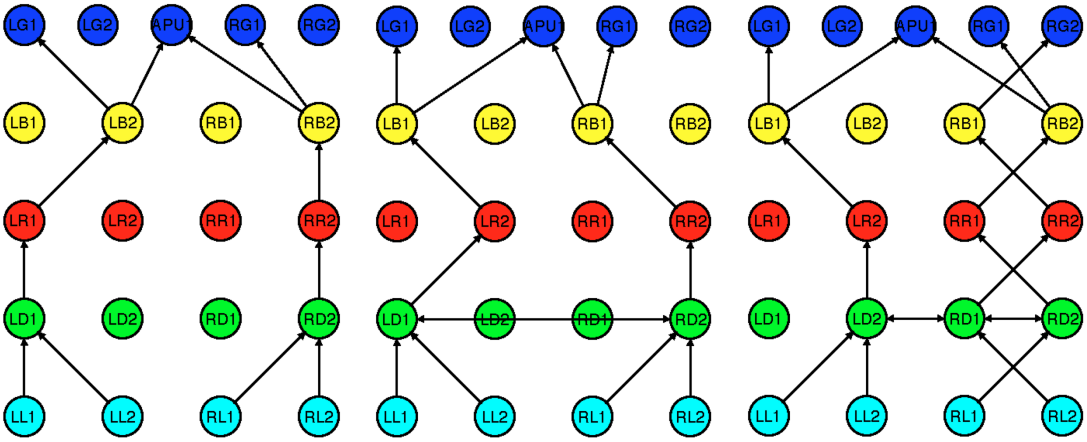
5.7.2 Optimization Results

We have developed ArchEx, a prototype framework for system architecture exploration and synthesis, implementing both the ILP-MR and ILP-AR algorithms. ArchEx leverages



(a) Architecture 1 (b) Architecture 2 (c) Architecture 3

Figure 5.2: Electric power system architectures and reliability as obtained at each iteration of an Integer Linear Programming Modulo Reliability (ILP-MR) run with $r^* = 2 \times 10^{-10}$: (a) $r = 6 \times 10^{-4}$; (b) $r = 2.8 \times 10^{-10}$; (c) $r = 0.79 \times 10^{-10}$.



(a) Architecture 1 (b) Architecture 2 (c) Architecture 3

Figure 5.3: Electric power system architectures synthesized using Integer Linear Programming with Approximate Reliability (ILP-AR) for different reliability requirements: (a) $r^* = 2 \times 10^{-3}, \tilde{r} = 6.0 \times 10^{-4}, r = 6 \times 10^{-4}$; (b) $r^* = 2 \times 10^{-6}, \tilde{r} = 2.4 \times 10^{-7}, r = 3.5 \times 10^{-7}$; (c) $r^* = 2 \times 10^{-10}, \tilde{r} = 7.2 \times 10^{-11}, r = 2.8 \times 10^{-10}$.

YALMIP [129] and CPLEX [5] to, respectively, formulate and solve ILP problems. All the numerical experiments were performed on an Intel Core i7, 2.3-GHz processor with 8-GB memory. Figure 5.2 shows the architectures obtained at each iteration of the ILP-MR algorithm for a load failure probability requirement $r^* = 2 \times 10^{-10}$. By solving for just the connectivity and power flow constraints, we obtain the simplest possible architecture (Fig-

ure 5.2a), which only provides a single path from a load to a generator (or APU), thus showing the highest failure probability. Based on the parameters in Table 5.1, we obtain $\rho = 8 \times 10^{-4}$, which leads to $k = 2$, as discussed in Section 5.6.1. Therefore, at the second iteration, two additional paths are enforced between each load and a generator, as shown in Figure 5.2 (b). Since the requirement is not yet satisfied, a third iteration is used to fine tune the reliability, by adding one more path between each load and an AC bus. The total computation time to generate the architectures in Figure 5.2 was about 38 s.

Three architectures, obtained using the ILP-AR algorithm for different load failure probability requirements, are, instead, shown in Figure 5.3. The lower the required failure probability, the higher the number of redundant paths and components instantiated from the original template, and the higher the associated cost. For each architecture, the approximate algebra provides an estimation $\tilde{r}$ of the failure probability which is extremely close to the actual value r obtained by exact computations. While the failure probability of the architecture in Figure 5.3 (c) exceeds the requirement, the error is well within the bound predicted by Theorem 5.4.1. The execution time of each optimization run in Figure 5.3 was also approximately 38 s; however, about 70% of the computation time was used to generate the optimization constraints, which can also be performed off-line for a given template.

To test the scalability of both the approaches, we designed EPS architectures with an increasing number of components. In Table 5.3, we report on the execution time of the ILP-MR approach using Algorithm 3 (at the top) in comparison with the one obtained by a lazier approach, which only enforce the addition of one additional path at each iteration between the load and a component with a minimal degree of redundancy. The dramatic reduction in time spent for reliability analysis (e.g., more than one day versus 3 min for a 50-node architecture) shows the advantage of using the analysis results to infer the number of required redundant paths, as proposed in Algorithm 3. When this inference is feasible, ILP-MR outperforms ILP-AR (see solver times in Table 5.4) for architectures with more than 40 nodes. On the other hand, as evident from Table 5.4, once the optimization problem is generated for a given template, ILP-AR is more competitive for smaller architectures. Yet, problems with several thousands of constraints, and including a realistic number of generators (normally less than 10), can still be formulated and solved in a few hours. We also observe that, because of the sparsity of the EPS adjacency matrix, in this case study, it was possible to reduce the number of generated constraints, which is always smaller than the asymptotic estimation in Section 5.5.

Overall, we infer that ILP-AR turns out to be preferable when we aim to a coarser estimation of the capability (and limitations) of an architecture template and a platform library in terms of reliability. On the other hand, ILP-MR makes it easier to incorporate domain-specific knowledge, since a designer can customize the techniques adopted to improve reliability at each iteration. Moreover, ILP-MR becomes the preferred choice, especially for larger problem instances, when we can estimate the number of redundant paths needed to satisfy the requirement as early as possible, or when we are willing to pay for a longer execution time to incrementally fine tune the reliability of the design.

Table 5.3: Number of iterations, reliability analysis and solver time for different electric power system architecture sizes ($r^* = 10^{-11}$, $n = 5$) using integer linear programming modulo reliability (ILP-MR) with LearnCons (top) and with a "lazier" strategy, enforcing only one additional path at each iteration (bottom).

$\|V\|$ (# Generators)	#Iterations	Analysis time (s)	Solver time (s)
20 (4)	3	34	4.3
30 (6)	3	78	9
40 (8)	3	106	14
50 (10)	3	181	18
20 (4)	4	72	13
30 (6)	7	852	28
40 (8)	10	9118	58
50 (10)	14	39563	114

Table 5.4: Number of constraints, problem generation (setup) and solver times for different electric power system architecture sizes ($r^* = 10^{-11}$, $n = 5$) using integer linear programming with approximate reliability (ILP-AR).

$\|V\|$ (# Generators)	# Constraints	Setup time (s)	Solver time (s)
20 (4)	5290	27	11
30 (6)	24514	402	77
40 (8)	74258	3341	494
50 (10)	176794	18902	5059

5.8 Conclusions

We provided a formal, graph-theoretical notion of cyber-physical system architecture, and showed how structural and topological contracts, which are relevant to architecture design, can be expressed as predicates over the nodes and edges of the architecture graph, and encoded as integer-linear constraints over a set of Boolean variables denoting the presence or absence of interconnections. We then formulated the architecture selection problem in terms of finding a mapping of an application contract (capturing system requirements) into an implementation contract (specifying the properties of the components and their interconnection) to minimize an overall cost (e.g. component number, weight). We finally cast this mapping problem as an integer-linear program (ILP).

Since generating exact reliability constraints by enumeration of failures on all possible graph configurations can take exponential time, we proposed an approximate measure that can be efficiently computed and provides reliability estimations to the correct order of magnitude, and with an explicit bound on the approximation error. Based on this measure, we introduced and characterized two efficient ILP-based algorithms for the optimal selection of system architectures subject to safety and reliability requirements. The Integer-Linear Pro-

gramming with Approximate Reliability (ILP-AR) algorithm generates larger, monolithic problem instances using efficient but approximate constraints computations; the Integer-Linear Programming Modulo Reliability (ILP-MR) algorithm breaks the complex optimization task into a sequence of smaller optimization tasks interleaved with exact reliability checks. By relying on efficient mechanisms to prune out large portions of the discrete space that are inconsistent with the reliability requirements, ILP-MR can outperform ILP-AR on large problem instances. We implemented both the algorithms in the ArchEx framework, and demonstrated their effectiveness on the design of aircraft power system architectures. The resulting architecture can be provided as part of the "specification" for the control design step in our methodology, as further detailed in Chapter 6.

Chapter 6

Contract-Based Control Design and Verification

This chapter introduces two paradigms for systematic, contract-based design of control strategies, which combine optimization-based mapping with pre-existing control design and synthesis techniques. The Reactive Synthesis-Based Optimized Control Mapping (RS-OCM) method enables the generation of hierarchical and distributed controller architectures by combining reactive synthesis from linear temporal logic contracts with optimization techniques based on contract refinement checking. The Programming-Based Optimized Control Mapping (P-OCM) method uses, instead, a formalization of the design requirements and the plant model in terms of arithmetic constraints over real numbers, and formulates the control problem as an optimization problem that is solved within a receding horizon approach to determine a correct control policy that can also optimize some performance metrics. The demonstration of an efficient algorithm for contract refinement checking concludes the chapter, showing how a contract library can be used to substantially accelerate verification and mapping tasks.

6.1 Reactive Synthesis-Based Optimized Control Mapping (RS-OCM): Overview

As mentioned in Section 2.2, *reactive systems* are systems that maintain an ongoing interaction with their environment by appropriately reacting to it. The controllers that regulate the behavior of such systems, which are considered in this thesis, are called *reactive controllers*. Given an architecture, e.g. resulting from the design methodology discussed in Chapter 5, we express the controller requirements as a contract $\mathcal{C}_C$: the assumptions A_C encode the allowable behaviors of the environment (including the physical plant) and the guarantees G_C encode the desired behaviors of the controller, which coincide with the desired behaviors of the closed-loop system.

In the Reactive Synthesis-Based Optimized Control Mapping (RS-OCM) paradigm, which is also pictured in Figure 4.2, we assume that $\mathcal{C}_C$ can be expressed as the conjunction be-

tween a contract $\mathcal{C}_{C,syn}$, for which the *reactive synthesis* problem is tractable, and a synthesis method can be efficiently applied, and a contract $\mathcal{C}_{C,ver}$, which is instead checked by a verification routine, which we call an *oracle*, in an optimization loop. While the paradigm can be, in principle, instantiated using different formalisms, in this thesis, we assume that $\mathcal{C}_{C,syn}$ is expressed using linear temporal logic (LTL), i.e. $\mathcal{C}_{C,syn} = \mathcal{C}_{LTL}$, while $\mathcal{C}_{C,ver}$ is expressed in signal temporal logic (STL), i.e. $\mathcal{C}_{C,ver} = \mathcal{C}_{STL}$. The oracle is a contract monitoring routine operating on simulation traces. The STL formulas in $\mathcal{C}_{STL}$ can either be obtained by heterogeneous refinement of a subset of LTL formulas in $\mathcal{C}_{LTL}$ or generated anew to capture design aspects related to the plant and the hardware implementation of the control algorithm, which cannot be expressed using the Boolean, untimed, or DE abstractions offered by LTL. $\mathcal{C}_{LTL} \wedge \mathcal{C}_{STL}$ is then a *vertical contract* for the controller, since $\mathcal{C}_{LTL}$ and $\mathcal{C}_{STL}$ refer to two different controller representations, possibly involving different viewpoints (e.g. functional and timing).

Example 25 (LTL/STL Vertical Contract)**.** *We consider a simple power network topology consisting of a bus B connected with generators G_1 and G_2 via contactors C_1 and C_2, respectively. Moreover, we assume the following requirement for the controller: "if G_1 fails and G_2 is healthy, then the controller shall first open C_1 and then close C_2, while guaranteeing that B does not lose power for more than t_{max}". We can encode this requirement as a conjunction of LTL and STL contracts as follows:*

- *Let g_i and c_i ($i = 1, 2$) be Boolean variables encoding the status of generators (healthy/unhealthy) and contactors (open/closed). Then, the LTL contract can be used to capture the desired sequence of actions prescribed by the requirement, i.e. $\mathcal{C}_{LTL} = (\{g_1, g_2, c_1, c_2\}, \mathtt{T}, \square\{\neg g_1 \wedge g_2 \rightarrow \neg c_1 \wedge (\bigcirc c_2)\})$.*
- *Furthermore, we can refine the "absence of power losses on a bus for more than t_{max}" with the statement: "the bus voltage V_B deviates from the desired value V_d by more than a margin ϵ for more than t_{max}", and use the following STL formula to state that the above faulty behavior should never happen: $\phi = \neg(\diamondsuit_{[0,\infty)} \square_{[0,t_{max}]}(|V_B(t) - V_d| \geq \epsilon))$. This translates into the contract $\mathcal{C}_{STL} = (\{V_B\}, \mathtt{T}, \phi)$.*

A brief introduction to the RS-OCM paradigm was already provided in Section 4.4.3. In this section, we detail the controller design steps used to guarantee the consistency of $\mathcal{C}_{C,syn} \wedge \mathcal{C}_{C,ver}$ and refine it towards an implementation: reactive synthesis and optimized mapping. Then, in Section 6.2, we describe an instance of the RS-OCM paradigm as applied to the design of an aircraft electric power system controller. To implement the RS-OCM methodology, we use a combination of tools from the computer science and formal methods domains, as also introduced in Section 2.7. Finally, numerical results from the application of the RS-OCM method to a concrete case study will be reported in Chapter 7.

6.1.1 Reactive Synthesis

In the RS-OCM paradigm, $\mathcal{C}_{LTL}$ is first used together with DE models of the plant components (also described by LTL formulas) to synthesize a reactive control protocol in the form of one (or more) state machines using reactive synthesis techniques from LTL specifications, as described in Section 2.7.1. The resulting high-level controller will satisfy $\mathcal{C}_{LTL}$ by construction. While the problem has a doubly exponential complexity for general LTL [163], the generalized reactivity (1) (GR(1)) fragment of LTL [160] generates problems that can be solved in polynomial time, i.e., polynomial in $|dom(E) \times dom(D)|$, the number of valuations of the variables in E and D, where E is the set of environment (input) variables and D is the set of controlled (output) variables of a DE controller.

Given the LTL specification $\varphi = (\varphi_e \rightarrow \varphi_s)$, where φ_e characterizes the assumptions on the environment and φ_s characterizes the controller (system) guarantees, GR(1) specifications restrict φ_e and φ_s to take the following form, for $\alpha \in \{e, s\}$,

$$\varphi_\alpha := \varphi^\alpha_{\text{init}} \wedge \bigwedge_{i \in I^\alpha_1} \Box \varphi^\alpha_{1,i} \wedge \bigwedge_{i \in I^\alpha_2} \Box \Diamond \varphi^\alpha_{2,i},$$

where $\varphi^\alpha_{\text{init}}$ is a propositional formula characterizing the initial conditions; $\varphi^\alpha_{1,i}$ are transition relations characterizing safe, allowable moves and propositional formulas characterizing invariants; $\varphi^\alpha_{2,i}$ are propositional formulas characterizing states that should be attained infinitely often; I^α_1 and I^α_2 are index sets enumerating formulas $\varphi^\alpha_{1,i}$ and $\varphi^\alpha_{2,i}$, respectively. In this chapter, we focus on synthesis of control protocols from contracts that can be expressed in GR(1), and build on recent works on formal synthesis of aircraft vehicle management systems [207], distributed control synthesis [157], and reactive synthesis for electric power systems [211]. In our framework, optimized mapping, e.g. performed via simulation, may be an expensive or incomplete procedure for certain kinds of requirements; reactive synthesis is then key to make it affordable, by guaranteeing that several functional, safety and reliability requirements are already satisfied by construction.

6.1.2 Distributed Synthesis

To provide an inherent level of redundancy for system reliability, distributed control architectures are increasingly being adopted in several safety-critical cyber-physical systems (CPS), thus motivating the extension of reactive synthesis techniques to the design of distributed controllers. Contracts can offer a natural framework to reason about distributed control architectures in a compositional way. In distributed synthesis, different control subsystems can be composed if their contracts are compatible. Hence, the goal of distributed synthesis is to simultaneously refine a system contract into compatible horizontal contracts for the components (i.e., subsystems), and to find the control logics that realize those contracts.

In a *top-down* approach, given a global specification and a system composed of subsystems, distributed synthesis proceeds by first finding local specifications for each subsystem,

and then synthesizing local controllers for these subsystems separately. If the local specifications satisfy certain conditions, it can be shown that the local controllers realizing these local specifications can be implemented together and the overall system is guaranteed to satisfy the global specification. Conversely, in a *bottom-up* approach, we assume that the local specifications are given in terms of a library of contracts that are realizable. Our goal is then to find an aggregation of contracts from the library that refines the global specification contract. A distributed (global) controller can also be built, in this phase, by composition from a library of pre-synthesized (local) controllers, using optimized mapping techniques. In this context, an optimization problem can be cast as in (4.1), where contract refinement checking is repeatedly performed in an optimization loop. Therefore, devising algorithms that can efficiently check contract refinement is key. We discuss one of such algorithms in Section 6.4.

For example, we illustrate below a special case of distributed architecture, i.e. a serial interconnection of controllers, which will be used in Chapter 7 to synthesize controllers for the AC and DC subsystems of an electrical power system separately. Noticeably, the following theorem is based on a previous result [157]. Yet, the new proof provided below shows that contracts offer straightforward mechanisms for rigorous and effective compositional reasoning in distributed control architectures.

Theorem 6.1.1. *Given*

- *a system characterized by a set $S = D \cup E$ of variables, where D and E are disjoint sets of controllable and environment variables,*
- *its two subsystems with variables $S_1 = D_1 \cup E_1$ and $S_2 = D_2 \cup E_2$, where for each $i \in \{1, 2\}$, D_i and E_i are disjoint sets of controllable and environment variables for the i^{th} subsystem, D_1 and D_2 are disjoint, and $D = D_1 \cup D_2$,*
- *a set ι of pairs of variables representing the interconnection structure, that is, for a serial interconnection, $\iota = \{(o_1, i_2) | o_1 \in O_1 \subseteq (D_1 \cup E_1), i_2 \in I_2 \subseteq E_2\}$, where for all $(o, i) \in \iota$, i is set equal to o (and, possibly, renamed as o),*
- *a global specification $\varphi : \varphi_e \rightarrow \varphi_s$, and two local specifications $\varphi_1 \colon \varphi_{e_1} \rightarrow \varphi_{s_1}$ and $\varphi_2 \colon \varphi_{e_2} \rightarrow \varphi_{s_2}$, where φ_e, φ_{e_1}, φ_{e_2}, φ_s, φ_{s_1}, and φ_{s_2} are LTL formulas containing variables only from their respective sets of environment variables E, E_1, E_2 and system variables S, S_1, S_2;*

if the following conditions hold:

1. *any behavior that satisfies φ_e also satisfies $(\varphi_{e_1} \wedge \varphi_{e_2})$,*
2. *any behavior that satisfies $(\varphi_{s_1} \wedge \varphi_{s_2})$ also satisfies φ_s,*
3. *there exist two controllers that make the local specifications $(\varphi_{e_1} \rightarrow \varphi_{s_1})$ and $(\varphi_{e_2} \rightarrow \varphi_{s_2})$ true under the interconnection structure ι;*

then, implementing the two controller together leads to a controller that satisfies the global specification $\varphi_e \rightarrow \varphi_s$.

Proof. The conditions on D, D_1, D_2 ensure that the two controllers are composable, i.e. they do not try to control the same output (controllable) variables. We first derive LTL contracts from the global and local specifications, by defining the following sets of behaviors in terms of assumptions and guarantees:

$$
\begin{aligned}
&A = \{\sigma : \sigma \models \varphi_e\}; \quad G = \{\sigma : \sigma \models (\varphi_e \rightarrow \varphi_s)\};\\
&A_i = \{\sigma : \sigma \models \varphi_{e_i}\}; \quad G_i = \{\sigma : \sigma \models (\varphi_{e_i} \rightarrow \varphi_{s_i})\};\\
&A' = \{\sigma : \sigma \models (\varphi_{e_1} \wedge \varphi_{e_2})\}; \quad G' = \{\sigma : \sigma \models ((\varphi_{e_1} \wedge \varphi_{e_2}) \rightarrow (\varphi_{s_1} \wedge \varphi_{s_2}))\},
\end{aligned}
$$

where all the behaviors are to be interpreted, after variable renaming, extension, and equalization, over the global alphabet[1].

Let $\mathcal{C} = (A, G)$ be the global contract, $\mathcal{C}_1 = (A_1, G_1)$ and $\mathcal{C}_2 = (A_2, G_2)$ the local contracts, and $\mathcal{C}' = (A', G')$, all in saturated form. We immediately observe that $A' = A_1 \cap A_2$ while $G' \supseteq (G_1 \cap G_2)$. Clearly, for any implementation M_i, $M_i \models \mathcal{C}_i$ if and only if its set of behaviors $[\![M_i]\!] \subseteq G_i$, i.e. $[\![M_i]\!] \models \varphi_i$, after alphabet equalization. Moreover, because any implementations M_1 and M_2 of $\mathcal{C}_1$ and $\mathcal{C}_2$ are composable, contract composition using equations (2.2) and (2.3) is well defined and the composition $M_1 \times_\iota M_2$ (under the interconnection ι) is an implementation of $\mathcal{C}_1 \overset{\iota}{\rightsquigarrow} \mathcal{C}_2 := \mathcal{C}_1 \otimes \mathcal{C}_2^\iota$ (under the interconnection ι). By condition 3 in the statement of the theorem, we know that such an implementation $M_1 \times_\iota M_2$ exists. Therefore, $\mathcal{C}_1 \otimes \mathcal{C}_2^\iota$ is compatible (and consistent).

To show the result, it is then enough to prove that $\mathcal{C}_1 \otimes \mathcal{C}_2^\iota \preceq \mathcal{C}$, i.e., $\mathcal{C}_1 \otimes \mathcal{C}_2^\iota = (A_{12}, G_{12})$ refines $\mathcal{C}$. By the definition of refinement, this amounts to showing that $G_{12} \subseteq G$ and $A_{12} \supseteq A$. This is straightforward, since

$$G_{12} = (G_1 \cap G_2) \subseteq G' \subseteq G \tag{6.1}$$

by conditions 1 and 2 in the theorem statement, and

$$A_{12} = (A_1 \cap A_2 \cup \neg G_{12}) \supseteq (A_1 \cap A_2) = A' \supseteq A, \tag{6.2}$$

by condition 1. Then, we conclude $\mathcal{C}_1 \otimes \mathcal{C}_2^\iota \preceq \mathcal{C}' \preceq \mathcal{C}$, and $M_1 \times_\iota M_2$ satisfies the global specification[2]. □

There are two sources of conservatism in distributed synthesis. The first one is due to the fact that local controllers have only local information. Therefore, even if there exists a

[1] We choose here to express contracts using a set-based notation. However, we can equivalently express them as pairs of formulas, as we do in Section 2.4.1.4.

[2] In this proof, we do not need to check contract compatibility by universal quantification over D, i.e. whether $\forall D : A_{12} \neq \emptyset$, since we are assured that $\mathcal{C}_1$ and C_2 are compatible under ι by condition 3 in the theorem statement.

centralized controller that realizes a global specification, there may not exist local controllers that do so. This is an inherent problem and can only be addressed by modifying the control architecture (e.g., by changing the mapping of controlled variables to controllers, by introducing new sensors, or by modifying the information flow between local controllers). The second source of conservatism is computational. Even when local controllers that realize the global specification exist, it might be difficult to find them (e.g., see Pnueli and Rosner [163] for some undecidability results). We note that the conditions provided in Theorem 6.1.1 are only sufficient conditions. The choices of φ_{e_j} and φ_{s_j} for $j \in \{1, 2\}$ plays a role in the level of conservatism. In principle, φ_{e_j} and φ_{s_j} should be chosen such that A' is as "small" as possible, and G' is as "large" as possible in the sense of set inclusion. Hence, when conditions 1 and 2 are satisfied but condition 3 is not satisfied, one can gradually refine the local specifications to make them realizable [157].

6.1.3 Optimized Mapping

Several real-time performance requirements (e.g. timing constraints), mostly relating to the dynamic behaviors of the physical plant and the hardware implementation of the control algorithm, cannot be effectively imposed via synthesis at the DE level. For this purpose, we refine a subset of LTL requirements into STL constructs on physical (e.g. electrical, mechanical) quantities and leverage off-line or on-line monitoring of simulation traces while optimizing the system.

The functional model of the centralized or distributed controller, synthesized using the techniques in Section 6.1.1 and Section 6.1.2, is then embedded into a high-fidelity model of the system, e.g. a hybrid model, possibly including an acausal representation of the plant. The joint execution of the controller with the plant in this mapping step is instrumental to: (i) check the consistency of the vertical contract $\mathcal{C}_{LTL} \wedge \mathcal{C}_{STL}$, (ii) discharge the timing assumptions made during the synthesis step, and (iii) ultimately verify the satisfaction of both the functional and timing viewpoints. The resulting optimal controller and plant configurations after mapping are returned as the final design. In the following, we discuss an instance of problem (4.1) in the case of STL contracts.

In this context, a Boolean verdict on whether a property is satisfied may not be sufficient for design space exploration and system optimization. In fact, we are also interested in capturing the *robustness of satisfaction* of a formula φ by a signal $\boldsymbol{q}$, i.e., the amount of margin by which a property is satisfied. To do so, we refer to the *quantitative semantics* of STL. The quantitative semantics of STL are defined using a real-valued function ρ of a trace $\boldsymbol{q}$, a formula φ, and time t satisfying the following property:

$$\rho(\varphi, \boldsymbol{q}, t) \geq 0 \text{ iff } \ (\boldsymbol{q}, t) \models \varphi. \tag{6.3}$$

The underlying idea is that, whenever the absolute value of $\rho(\varphi, \boldsymbol{q}, t)$ is large, a change in $\boldsymbol{q}$ is less likely to affect the Boolean satisfaction (or violation) of φ by $\boldsymbol{q}$, i.e. the margin by which a design satisfies φ is larger.

Without loss of generality, an STL predicate μ can be identified to an inequality of the form $g(\boldsymbol{q}) \geq 0$ (the use of strict or non strict inequalities is a matter of choice and other inequalities can be trivially transformed into this form). From this form, a straightforward quantitative semantics for predicate μ is defined as

$$\rho(\mu, \boldsymbol{q}, t) = g(\boldsymbol{q}(t)). \tag{6.4}$$

Then ρ can be inductively defined for every STL formula using the following rules:

$$\rho(\neg\varphi, \boldsymbol{q}, t) = -\rho(\varphi, \boldsymbol{q}, t) \tag{6.5}$$

$$\rho(\varphi_1 \wedge \varphi_2, \boldsymbol{q}, t) = \min(\rho(\varphi_1, \boldsymbol{q}, t), \rho(\varphi_2, \boldsymbol{q}, t)) \tag{6.6}$$

$$\rho(\varphi_1 \mathcal{U}_I \varphi_2, \boldsymbol{q}, t) = \sup_{t' \in t+I} \Big[\min \big(\rho(\varphi_2, \boldsymbol{q}, t'), \inf_{t'' \in [t, t')} \rho(\varphi_1, \boldsymbol{q}, t'') \big) \Big]. \tag{6.7}$$

Additionally, by combining equation (6.7), and $\Box_I \varphi \triangleq \neg \Diamond_I \neg \varphi$, we get

$$\rho(\Box_I \varphi, \boldsymbol{q}, t) = \inf_{t' \in t+I} \rho(\varphi, \boldsymbol{q}, t'). \tag{6.8}$$

Finally, for $\Diamond$, we get a similar expression using sup instead of inf. It can be shown that ρ, as defined above, satisfies equation (6.3) and thus defines a quantitative semantics for STL [76].

By leveraging such quantitative semantics, a design space exploration problem on a hybrid system model, defined as in Section 2.4.1.2, can be formulated as follows. Let $\mathcal{C}_{STL} = (\varphi_e, \varphi_e \rightarrow \varphi_s)$ be an STL contract encoding a set of system requirements, with φ_e and φ_s Parametric STL (PSTL) formulas. Let $\boldsymbol{C}$ be an array of costs, and $\boldsymbol{\kappa} \in \mathcal{K}$ a vector of platform configuration parameters, i.e., a vector of variables in the hybrid system model that are selected as a result of the design process. Our goal is to find a set of parameter vectors $\boldsymbol{\kappa^*}$ that are Pareto optimal with respect to the objectives in $\boldsymbol{C}$, while guaranteeing that the system satisfies φ_s for all possible system traces $\boldsymbol{s} \in \mathcal{S}$ satisfying the environment assumptions φ_e. Examples of design parameters could be the controller clock or a tunable delay in a component.

To formalize the above multi-objective optimization problem, we partition φ_s as

$$\varphi_s(\boldsymbol{\tau}, \boldsymbol{\pi}) = \varphi_{sc}(\boldsymbol{\tau}, \boldsymbol{\pi}) \wedge \bigwedge_{i=1}^{m} \varphi_{sr,i}(\boldsymbol{\tau}, \boldsymbol{\pi}), \tag{6.9}$$

where a set of time parameters $\boldsymbol{\tau} \in \mathcal{T}$ and scale parameters $\boldsymbol{\pi} \in \mathcal{P}$ can be used to capture degrees of freedom that are available in the system specifications, and whose final value can also be determined as a result of the optimization process. The formula φ_{sc} in (6.9) encodes the requirements that will be considered as "hard" optimization constraints for Boolean satisfaction, while $\varphi_{sr,i}$ are formulas that will also be considered for robust satisfaction, i.e., given a system trace $\boldsymbol{s'}$ and a parameter set $(\boldsymbol{\tau'}, \boldsymbol{\pi'})$, the robust satisfaction $\rho_i(\varphi_{sr,i}(\boldsymbol{\tau'}, \boldsymbol{\pi'}), \boldsymbol{s'}, 0)$ will also be computed. Similarly, the array of costs $\boldsymbol{C}$ can be partitioned as follows

$$\boldsymbol{C}(\boldsymbol{\kappa},\boldsymbol{\tau},\boldsymbol{\pi}) = \left(\boldsymbol{C}_c(\boldsymbol{\kappa},\boldsymbol{\tau},\boldsymbol{\pi}), C_i(\rho_i(\varphi_{s,ri}(\boldsymbol{\tau},\boldsymbol{\pi}),\boldsymbol{s}(\boldsymbol{\kappa}),0))\Big|_{1\le i\le m}\right), \tag{6.10}$$

where $\boldsymbol{C}_c(\boldsymbol{\kappa},\boldsymbol{\tau},\boldsymbol{\pi})$ is a vector of costs that depend only on the parameters of the model and the formulas; it can be used to capture, for instance, some performance figures (e.g., bandwidth, energy) as a function of the system design parameters, or the duration of a requirement violation. Each component $C_i(\rho_i(\varphi_{sr,i},\boldsymbol{s},0))$ in (6.10) is instead a scalar function of the quantitative satisfaction of each formula $\varphi_{sr,i}$; it can be used to capture and maximize the margin by which $\varphi_{sr,i}$ is satisfied.

By putting it all together, the design exploration problem can be expressed as a multi-objective robust optimization problem

$$\begin{aligned} \min_{\boldsymbol{\kappa}\in\mathcal{K},\boldsymbol{\tau}\in\mathcal{T},\boldsymbol{\pi}\in\mathcal{P}} \quad & \boldsymbol{C}(\boldsymbol{\kappa},\boldsymbol{\tau},\boldsymbol{\pi}) \\ \text{s.t.} \begin{cases} \mathcal{F}(\boldsymbol{s},\boldsymbol{\kappa}) = 0 \\ \\ \boldsymbol{s} \models \varphi_s(\boldsymbol{\tau},\boldsymbol{\pi}) \qquad \forall \boldsymbol{s} \quad \text{s.t.} \quad \boldsymbol{s} \models \varphi_e(\boldsymbol{\tau},\boldsymbol{\pi}) \end{cases} \end{aligned} \tag{6.11}$$

where we aim to minimize a set of costs over all possible system and formula parameter valuations, for all the system behaviors satisfying the behavioral model and the contract $\mathcal{C}_{STL}$. For a given parameter valuation $\boldsymbol{\kappa'}$, $\boldsymbol{s'} = (\boldsymbol{u'},\boldsymbol{y'},\boldsymbol{x'})$ is the trace of input, output and internal signals that are obtained by simulating $\mathcal{F}(.)$. A multi-objective optimization algorithm with simulation in the loop can then be used to find the Pareto optimal solutions $\boldsymbol{\kappa}^*$. While this may be expensive in general, it becomes affordable in many practical cases, as further illustrated in Section 6.2 and Chapter 7.

6.2 Reactive Synthesis-Based Optimized Control Mapping: Power System Design Example

We apply the RS-OCM methodology to the design of the Bus Power Control Unit (BPCU) of an aircraft electric power system (EPS). The formulation in this section will be used for the numerical experiments in Chapter 7.

Power requirements of different loads might differ in an aircraft based on the mode of operation. Similarly, the availability of the generators and the health conditions of several components might vary during the flight. The goal of the controller is to reconfigure the electric power system and reroute power by appropriately reacting to such changes in system conditions to ensure that safety-critical loads are always powered according to their safety and reliability requirements. We first describe how the control logic for the BPCU can be automatically synthesized from LTL contracts. Then, we discuss the use of STL and simulation-based design space exploration to check or enforce real-time constraints (e.g. timing) for controller implementation.

6.2.1 Synthesis of Reactive Protocols for Electric Power Distribution

The control protocol synthesis problem for electric power system can be stated as follows: given an electric power system topology (generated as discussed in Chapter 5) and a formal specification describing assumptions on the plant components and the external environment and guarantees on the system, build a controller that reconfigures the system (via turning on and off the contactors) by sensing and reacting to the faults and the changes in system status so as to ensure that the specification is met. Next, we discuss how to formalize the requirements to recast the above problem as a reactive synthesis problem.

6.2.1.1 Variables

Environment variables include the health statuses of components that are uncontrolled. In our formulation, we consider only generators, APUs, and rectifier units as environment variables. They can each take values of healthy (1) and unhealthy (0), and may change at any point in time[3]. *Controlled* variables are contactors, and can each take values of open (0) or closed (1). A closed contactor allows power to pass through, while an open one does not. *Dependent* variables are buses that can be either powered (1) or unpowered (0). Bus values will depend on the status of their neighboring contactors, buses, as well as the health status of connecting generators, APUs, or rectifier units.

Timing considerations play a key part in the specifications for an electric power system. LTL, however, only addresses the notion of temporal ordering of events. To reconcile this discrepancy, we handle timing annotations by introducing clock variables. Verification of actual timing constraints related to the controller implementation is then performed at a lower abstraction level, as detailed in Section 6.1.3. The following lists the temporal logic formulas used to concretely express the contract for controller synthesis for the primary distribution in an electric power system.

6.2.1.2 Environment Assumptions

Let $\mathcal{I}$ be an index set enumerating the set of environment variables described in Section 6.2.1.1. For each environment variable e_i, $i \in \mathcal{I}$, let p_i be its probability of failure in a given time interval T (e.g., the duration of a flight), as also used in Section 5.7 (footnote). Let r_S be the overall reliability level the system has to achieve, that is, the probability of the overall system failure should be less than or equal to r_S. Assuming independence of component failures, the overall reliability level determines the allowable environment assumptions by providing a bound on the number of simultaneous component failures allowed. More formally, denote a single configuration of the environment (i.e., an environment state) by $\mathbf{e}$. For a given subset $\mathcal{I}' \subseteq \mathcal{I}$ of the environment variables, we define $\mathbf{e}_{\mathcal{I}'} = (e_1, \ldots, e_{|\mathcal{I}|})$, where $e_i = 0$

[3]Generators can be taken offline by the pilot or may stop functioning due to a fault. We do not differentiate between these cases and simply call a generator unhealthy when it is unavailable or malfunctioning.

(unhealthy) if $i \in \mathcal{I}'$; and $e_i = 1$ (healthy) otherwise. Let $h : [0,1] \to 2^{2^{\mathcal{I}}}$ be the function that maps the system reliability level to the possible environment configurations. We can then enumerate all allowable environment configurations based on the required reliability level as

$$\mathcal{E}_S = \{\mathbf{e}_{\mathcal{I}'} | \mathcal{I}' \in h(r_S)\}. \tag{6.12}$$

With this definitions, an environment assumption can be written in LTL as $\square(\mathbf{e} \in \mathcal{E}_S)$.

As the function h can be difficult to compute, alternatively, one can reason about the probability r_C of an environment configuration and map it back to the system reliability level r_S. To this effect, we enumerate all environment configurations that occur with probability more than a given level r_C. Then, if the control synthesis problem is realizable with the assumption $\square(\mathbf{e} \in \mathcal{E}_C)$, this implies that the system level reliability is

$$r_S = \sum_{\mathbf{e} \notin \mathcal{E}_C} \prod_{j:e_j=0} p_j \prod_{j:e_j=1} (1 - p_j).$$

The second environment assumption is also related to failure analysis. We assume that when a component fails during the flight (the interval T), it will not come back online. This can be expressed in LTL as

$$\square \bigwedge_{i \in \mathcal{I}} ((e_i = 0) \to \bigcirc(e_i = 0)). \tag{6.13}$$

6.2.1.3 Controller Guarantees

We provide below the list of LTL guarantees that are part of the controller contract $\mathcal{C}_C$. The first set of formulas encode the DE behavioral model of the power buses in the system. The other formulas capture, instead, the system requirements. Any controller implementation should satisfy the conjunction of both of them.

Power Status of Buses: An AC bus can only be powered if there exists a *live path* (i.e., all contactors closed along a path) that connects the bus to a healthy AC generator or a healthy APU. Similarly, a DC bus can only be powered if there exists a live path that connects it to a healthy rectifier unit, which itself is connected to a powered AC bus. Let $\tilde{p}_{i,B}$ denote the set of all components (i.e., contactors and buses) along a path between bus B and environment variable e_i for $i \in \mathcal{I}$, excluding B and e_i. Furthermore, let $\mathcal{G} \subseteq \mathcal{I}$ and $\mathcal{R} \subseteq \mathcal{I}$ represent the sets of generators and rectifier units. AC bus B is powered if there exists a live path between B and e_i for $i \in \mathcal{G}$, written as[4]

$$\square \left\{ \bigvee_{i \in \mathcal{G}} \left((e_i = 1) \wedge \bigwedge_{X \in \tilde{p}_{i,B}} (x = 1) \right) \to (b = 1) \right\}. \tag{6.14}$$

[4]Per abuse of notation, we denote components by uppercase letters (e.g., C, B) and component statuses by lowercase letters (e.g., c, b).

If there exists no live path between B and a generator e_i for $i \in \mathcal{G}$, then B will be unpowered

$$\square \left\{ \neg \bigvee_{i \in \mathcal{G}} \left((e_i = 1) \wedge \bigwedge_{X \in \tilde{p}_{i,B}} (x = 1) \right) \rightarrow (b = 0) \right\}. \tag{6.15}$$

A similar set of specifications for DC buses holds in which environment variables e_i span $i \in \mathcal{R}$.

Balanced Power Flow in Nominal Conditions: Under nominal conditions (i.e., when all generators and rectifier units are healthy), the power drawn from each generator by the buses connected to it should be less than the capacity of that generator. Let $\tilde{P}_B$ be a constant that corresponds to the maximum power required by the loads connected to the bus B and $\tilde{P}_{e_i}$ be a constant corresponding to the power generator i can nominally provide. Using the live path constructs, we define the power variables $l_{i,B} \in \{0, \tilde{P}_B\}$ such that $\bigwedge_{X \in \tilde{p}_{i,B}} (x = 1) \rightarrow (l_{i,B} = \tilde{P}_B)$, and $\neg \bigwedge_{X \in \tilde{p}_{i,B}} (x = 1) \rightarrow (l_{i,B} = 0)$. Then, the power flow requirement can be written as

$$\square \left\{ \bigwedge_{i \in \mathcal{I}} (e_i = 1) \rightarrow \bigwedge_{i \in \mathcal{G}} (\tilde{P}_{e_i} \geq \sum_{B \in \mathcal{B}} l_{i,B}) \right\}, \tag{6.16}$$

where $\mathcal{B}$ represents the set of buses.

No Paralleling of AC Sources: To avoid paralleling, we explicitly enumerate and disallow all bad configurations. In Fig. 1.3, paralleling can occur if there exists a live path that connects two AC generators or APUs. Let $\tilde{p}_{i,j}$ represent the set of components along a path between generators e_i, e_j, for $i, j \in \mathcal{G}$ and $i \neq j$. We disallow configurations in which all contactors $C \in \tilde{p}_{i,j}$ create a live path. These specifications are written as

$$\square \bigwedge_{i,j \in \mathcal{G}} \left\{ \neg \bigwedge_{C \in \tilde{p}_{i,j}} (c = 1) \right\}. \tag{6.17}$$

Safety-Criticality of Buses: A safety-critical bus can be unpowered for no longer than T_s time steps. This is implemented through the use of an additional clock variable x_B for each bus B, where each "tick" of the clock represents δ time. If the bus is unpowered, then at the next time step clock x_B increases by δ. If B is unpowered, then at the next time step clock x_B resets to zero. Then, we limit the number of steps B can remain unpowered in

order to ensure that x_B never becomes larger than T_s. Thus, for all safety-critical buses[5],

$$\square \{(b = 0) \rightarrow (\bigcirc x_B = x_B + \delta)\}, \tag{6.18a}$$
$$\square \{(b = 1) \rightarrow (\bigcirc x_B = 0)\}, \tag{6.18b}$$
$$\square(x_B \leq T_s). \tag{6.18c}$$

Unhealthy Sources: A bus connected to an unhealthy source (generator or rectifier unit) will create a short-circuit failure, leading to excessive electrical currents, overheating, and possible fires. While generators have internal protections to avoid such failures, we require that appropriate contactors open when a generator or APU becomes unhealthy to isolate the unhealthy source and prevent its use. Let $\mathcal{N}(e_i)$ represent the set of contactors directly connected, or neighboring, environment variable e_i for $i \in \mathcal{I}$. We write the specifications to disconnect all unhealthy sources as

$$\square \bigwedge_{i \in \mathcal{I}} \left\{ (e_i = 0) \rightarrow \bigwedge_{C \in \mathcal{N}(e_i)} (c = 0) \right\}. \tag{6.19}$$

Both the assumptions and guarantees mentioned above form the controller contract $\mathcal{C}_C$. Moreover, since they are within the GR(1) fragment of LTL, digital synthesis tools, such as the one implemented in JTLV [165], can be used to automatically synthesize the control protocol. For the examples discussed in this thesis, we used the Temporal Logic Planning (TuLiP) Toolbox [209], a collection of PYTHON-based code for automatic synthesis of embedded control software, which provides an interface to JTLV. In Chapter 7 we show how the formulas above can be generated from a set of higher-level patterns, as a part of a domain specific language for power system distribution.

6.2.1.4 Capturing Actuation Delays

In the discussion above, we assumed ideal contactors that can be instantaneously controlled. It is possible to capture delays in contactor opening and closing times, as well as the communication delays between the controller and the contactors. To this effect, one can introduce a controlled variable $\tilde{C}$ to represent the controller intent for contactor C and treat the contactor as an environment variable. The uncertain delay between the controller intent and contactor state can be handled by the use of an additional clock variable x_C for each contactor C, where each "tick" of the clock represents δ time. If the contactor intent is open and the contactor state is closed, the contactor opens within $[T_{o_{min}}, T_{o_{max}}]$ units of time unless a close command is issued before it opens. If the contactor intent is closed and the contactor

[5] As apparent from the formulas above, we extend LTL with first order atoms including linear arithmetic constraints on integers. Moreover, in (6.18), we use a "functional" notation for the $\bigcirc$ operator, where $\bigcirc x_B$ is the value of the variable x_B at the next reaction index. This is a notation broadly used in model checkers, such as NUSMV.

state is open, the contactor closes within $[T_{c_{min}}, T_{c_{max}}]$ units of time unless an open command is issued before it closes. Once the contactor intent is set, if the contactor state does not match the intent, at the next step clock x_C will increase by δ. If contactor state and intent match, then at the next step clock x_C resets to zero:

$$\square\{(\bigcirc c = \tilde{c}) \rightarrow (\bigcirc x_C = 0)\}.$$

When the control command is the same as the contactor state, the contactor state remains the same, i.e.,

$$\square\{(\tilde{c} = c) \rightarrow (\bigcirc c = c)\}.$$

Finally, the assumptions capturing the contactor closing behavior in relation to the controller input intent are given by

$$\begin{aligned}
&\square\left\{(\tilde{c} = 1 \wedge c = 0 \wedge (x_C < T_{c_{min}})) \rightarrow (\bigcirc c = 0 \wedge \bigcirc x_C = x_C + \delta)\right\},\\
&\square\left\{(\tilde{c} = 1 \wedge c = 0 \wedge (x_C \geq T_{c_{min}})) \rightarrow (\bigcirc c = 1 \vee \bigcirc x_C = x_C + \delta)\right\},\\
&\square(x_C \leq T_{c_{max}}).
\end{aligned}$$

The contactor opening behavior can be formally captured in a similar manner. The formulas mentioned in this section enter to the control synthesis problem as new environment assumptions when delays are taken into account. It should also be noted that unhealthy sources can only be disconnected with a delay in this case, therefore formula (6.19) should be adjusted accordingly.

6.2.2 Simulation-Based Design Space Exploration

The design steps in Chapter 5 (Section 5.7) and Section 6.2.1 allow synthesizing electric power system architectures and control protocols that jointly satisfy the top-level system specifications, represented by contracts $\mathcal{C}_T$, for the EPS topology, and $\mathcal{C}_{C,LTL}$. To assess the satisfaction of real-time performance constraints, we monitor STL formulas from the controller contracts $\mathcal{C}_{C,STL}$ on voltage and current waveforms over time, as discussed in Section 6.1.3.

As an example, we investigate here the maximum reaction time allowed to the controller. For this purpose, we assume a synchronous implementation of the controller, running at a fixed period T_r[6]. In our hybrid model, the BPCU is connected in closed loop with the power system plant, while failure events can be injected by setting the input signal $\boldsymbol{u}(t)$. Moreover, we assume that all contactors respond with a fixed delay T_d to the open/close commands from the BPCU. We then consider the requirement that a DC essential bus must never be unpowered for more than t_{max} under any possible failure scenario. In a continuous

[6]The synchronous operation of the controller is a design choice specific to the case study in this thesis; it is not meant to serve as a general design guideline.

setting, such a requirement is translated by stating that the DC bus voltage V_{DC} should never deviate from the desired value V_d by more than a margin ϵ for more than t_{max}. The predicate specifying that the current value of the voltage stays in the desired range is: $|V_{DC}(t) - V_d| < \epsilon$. Then, the STL formula expressing this to be false for at least t_{max} is:

$$\chi = \square_{[0,t_{max}]} \neg(|V_{DC}(t) - V_d| < \epsilon). \tag{6.20}$$

Since we need to enforce that V_{DC} is never out of range only after the initial start-up transient time τ_i, we require

$$\phi(\tau_i) = \neg(\diamondsuit_{[\tau_i,\infty)}\ \chi) \tag{6.21}$$

to be true.

To compute the maximum amount of time elapsed while the DC bus voltage is out of range, i.e. for how long at most the voltage requirement on the DC bus is violated, we turn (6.20) into a PSTL formula, by introducing the timing parameter τ_e, after which an out-of-range voltage event is detected, as follows:

$$\psi(\tau_e) = \square_{[0,\tau_e]} \neg(|V_{DC}(t) - V_d| < \epsilon). \tag{6.22}$$

The initial start-up transient time τ_i is estimated from simulation as a function of T_r and T_d. Then, the maximum violation period $\tau_e^*(T_r, T_d)$ can be computed as the

$$\sup\{\tau_e \geq 0 \mid \phi\left(\tau_i(T_r, T_d), \tau_e\right) = \mathtt{F}\}, \tag{6.23}$$

where

$$\phi(\tau_i, \tau_e) = \neg(\diamondsuit_{[\tau_i,\infty)}\ \psi(\tau_e)). \tag{6.24}$$

The formula in (6.24) allows exploring the T_r-versus-T_d design space and finding the maximum allowed controller reaction time T_r^* for a fixed T_d^*, in such a way that the essential DC bus is never out of range for more than t_{max}. To do so, we cast an optimization problem following the formulation in (4.1)

$$\begin{aligned} &\min_{T_r>0} \quad 1/T_r \\ &\text{s.t.} \begin{cases} \mathcal{F}(\boldsymbol{u}, V_{DC}, T_r) = 0 \\ \\ V_{DC} \models \phi\left(\tau_i(T_r, T_d^*), \tau_e\right) \quad \forall \tau_e \geq t_{max} \quad \forall \boldsymbol{u} \text{ s.t. } \boldsymbol{u} \models \varphi_e' \end{cases} \end{aligned} \tag{6.25}$$

where $\boldsymbol{C} = 1/T_r$ is the cost function, $\boldsymbol{\kappa} = T_r$ is the design parameter, $\varphi_s(\boldsymbol{\tau}) = \bigwedge_{\tau_e \geq t_{max}} \phi\left(\tau_i(T_r, T_d^*), \tau_e\right)$ is the conjunction of PSTL formulas that must be satisfied, each parametrized by $\boldsymbol{\tau} = T_r$, and φ_e' refines the environment assumption formula φ_e in Section 6.2.1. In this case, the system behavior $\boldsymbol{s}$ is the trace $\boldsymbol{s} = (\boldsymbol{u}, V_{DC})$, where V_{DC} is the output signal to be observed during simulation, and $\boldsymbol{u}$ spans the set of all admissible failure injection traces that are consistent with the environment assumptions in Section 6.2.1.2.

The formulation in (6.25) includes an infinite set of formulas that must be satisfied for all admissible failure traces and values of $\tau_e \geq t_{max}$. However, such formulation can be further simplified, by observing that (6.25) is equivalent to

$$\begin{aligned} &\max_{T_r>0} \quad T_r \\ &\text{s.t.} \begin{cases} \mathcal{F}(\boldsymbol{u}, V_{DC}, T_r) = 0 \\ \\ \tau_e^*(T_r, T_d^*) \leq t_{max} \qquad \forall \boldsymbol{u} \text{ s.t. } \boldsymbol{u} \models \varphi_e' \end{cases} \end{aligned} \tag{6.26}$$

where τ_e^* is defined in (6.23). Moreover, as we will show in Chapter 7, it is enough to compute $V_{DC}(t)$ and τ_e^* under the worst case failure scenario, rather than for all possible failure traces, whenever the worst case assumptions on $\boldsymbol{u}(t)$ can be determined *a priori*. Problem (6.26) can then be solved by first solving the optimization problem in (6.23) to compute τ_e^* as a function of T_r and T_d^* in the worst case input scenario, and then by computing the value T_r^* of the controller reaction time that makes τ_e^* equal to t_{max}. For the example discussed in Chapter 7, we used the BREACH toolbox [77] to facilitate post-processing of simulation traces and verify the satisfaction of the STL formulas.

6.3 Programming-Based Optimized Control Mapping (P-OCM): Overview

The Programming-Based Optimized Control Mapping (P-OCM) method allows designing model predictive control algorithms [84] within a rigorous contract-based approach.

We leverage a *discrete-time abstraction* of the continuous behaviors of the system, and express contracts using either first-order difference equations involving the components' variables and parameters (e.g. for time-varying properties), or arithmetic constraints on real variables that must hold at each time step or at steady state (e.g. for time-invariant properties). The algebra of contracts can then be implemented by simply combining constraints via conjunction or disjunction to express, respectively, intersections or unions of behaviors.

Similarly to requirements (properties), component models can also be expressed at the same discrete-time abstraction level, i.e. by using difference equations to construct behavioral models, and arithmetic constraints on real numbers (e.g. polynomial constraints) for performance and cost models. An optimal control problem can then be formulated as an optimized mapping problem, an instance of (4.1), by "unrolling" the system dynamics over a time horizon H. Our goal is to find the value of the control law (a discrete-time continuous-valued trajectory or trace) subject to the aggregate component contracts, describing the system dynamics, and a top-level system contract, formalizing the top-level requirements, while minimizing an objective function. Such a formulation, e.g. generating a mixed integer-linear (or integer-quadratic) program to be solved in a receding horizon fashion, is returned as the final design. A pictorial representation of the control design flow is shown in Figure 6.1.

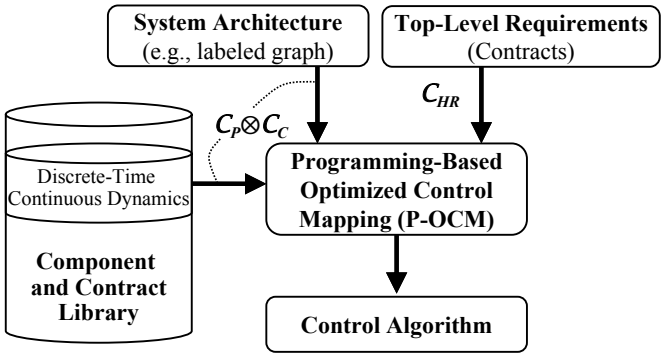


Figure 6.1: Programming-based optimized control mapping flow.

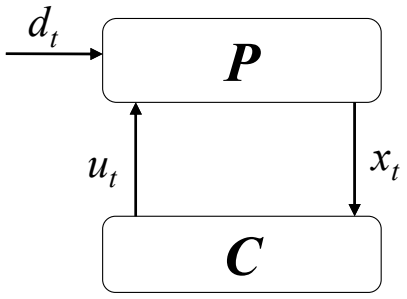


Figure 6.2: Generic feedback control scheme.

Formally, we refer to Figure 6.2, showing a generic feedback control scheme. In our contract-based framework, both the plant P and the controller C are specified as an aggregation of contracts from the library $\mathcal{L}$. To simplify, we denote the composition of the plant and controller contracts, under feedback interconnection and renaming, as $\mathcal{C}_P \otimes \mathcal{C}_C$. The top-level requirements are specified by a system-level application contract $\mathcal{C}_{RH}$. The refinement (mapping) between $\mathcal{C}_{RH}$ and $\mathcal{C}_P \otimes \mathcal{C}_C$ is then modeled as the *vertical contract* $\mathcal{C}_{RH} \wedge (\mathcal{C}_P \otimes \mathcal{C}_C)$ given by the *conjunction* of the architecture and application contracts. We are interested in an optimal control law subject to the constraint that $\mathcal{C}_{RH} \wedge (\mathcal{C}_P \otimes \mathcal{C}_C)$ is *consistent*, i.e. there exists an implementation satisfying both the guarantees of $\mathcal{C}_{RH}$ and $\mathcal{C}_P \otimes \mathcal{C}_C$ in the context of their assumptions, as further detailed below.

We assume that the aggregate system dynamics are described by the following difference

equation at each non-negative time step t:

$$x_{t+1} = p(x_t, u_t, d_t) \quad \forall\ t \in \mathbb{N}, \quad \forall\ x_0 \in \mathcal{X}_0, \quad \forall\ d(t) \in \mathcal{D}, \tag{6.27}$$

where x_t is the system state, u_t the control input, and d_t an external uncontrolled input (disturbance), all at time t. $\mathcal{X}_0$ is the set of all admissible initial conditions, while $\mathcal{D}$ is the set of all the admissible environment behaviors (traces). We denote as $S = X \cup U \cup D$ the set of system variables. A system behavior or trajectory $\sigma = s_0 s_1 s_2 \ldots$ is a sequence of valuations $s = (x, u, d)$ over S for all $t \in \mathbb{N}$. We also use $s(t)$ to denote a system behavior (trace) to distinguish it from from the valuation s_t at time t. The difference equation in (6.27) encodes the guarantees of the aggregate system contract $\mathcal{C}_P \otimes \mathcal{C}_C$, while the legal sets of input condition and environment trajectories encode the assumptions.

In general, a dynamical model for the system guarantees, as the one in (6.27), can be obtained by composition of the components' dynamics according to the system interconnection structure, e.g. by conjoining the components' dynamics, after extending their local variable alphabets to a global, common set of symbols. Such a conjunction may also include "local," component-level constraints, both in the form of difference equations or stateless (memoryless) constraints. In fact, the behavior of a component in P or C is guaranteed, according to its behavioral model, only under a set of constraints expressing its assumptions. Some of these constraints, e.g. including bounds on the magnitude of some of the system variables, must be discharged by the guarantees of other components, and explicitly accounted for in the optimization problem. To simplify, we model this kind of assertions in terms of stateless constraints of the form

$$\lambda_t(s_t) \leq 0 \quad \forall\ t \in \mathbb{N}, \tag{6.28}$$

where $\lambda_t(.)$, at each t, is a real-valued function of the system variables s.

We then assume that $\mathcal{C}_{HR}$ has the same assumptions of $\mathcal{C}_P \otimes \mathcal{C}_C$ in terms of admissible external inputs and initial conditions. Moreover, $\mathcal{C}_{HR}$ may have additional guarantees capturing the system-level requirements. To be concrete, we assume these guarantees can be expressed in term of stateless constraints as follows:

$$G_{HR} := \gamma_t(s_t) \leq 0 \quad \forall\ t \in \mathbb{N}, \tag{6.29}$$

where $\gamma_t(.)$, at each t, is also a real-valued function of the system variables s. However, expressions depending on the history of s rather than just s_t are also possible.

Finally, let $J(u_t, x_t)$ be a real function providing the system cost in terms of system state and control at time t. Then, the optimal control problem aiming at minimizing the cost over time horizon H while satisfying the system dynamics and contracts can be formulated as

follows:

$$\min_{\boldsymbol{u}_t} \quad \sum_{k=0}^{H-1} J(u_{t+k}, x_{t+k}) \tag{6.30a}$$

$$\text{subject to:} \quad x_{t+k+1} = p(x_{t+k}, u_{t+k}, d_{t+k}), \quad \forall\, k \in \{0, ..., H-1\} \tag{6.30b}$$

$$\gamma^i_{t+k}(x_{t+k}, u_{t+k}, d_{t+k}) \leq 0, \quad \forall\, k \in \{0, ..., H-1\}, \quad \forall\, i \in \{1, ..., |\Gamma|\} \tag{6.30c}$$

$$\lambda^j_{t+k}(x_{t+k}, u_{t+k}, d_{t+k}) \leq 0, \quad \forall\, k \in \{0, ..., H-1\} \quad \forall j \in \{1, ..., |\Lambda|\} \tag{6.30d}$$

$$x_t \in \mathcal{X}_t \tag{6.30e}$$

$$d_{t+k} \in \mathcal{D}_{t+k}, \quad \forall\, k \in \{0, ..., H-1\} \tag{6.30f}$$

where $\boldsymbol{u}_t = (u_t, u_{t+1}, \ldots, u_{t+H-1})$ is the control trajectory, Γ is the set of global (application) constraints as in (6.29), and Λ is the set of local (architecture) constraints as in (6.28). The resulting optimal control algorithm executes the optimization problem (6.30) in a receding horizon fashion to find the final $\boldsymbol{u}^*_t$.

A concrete example of this approach, including both formulation and numerical results, will be provided in Chapter 7, where we propose HOLMS, a hierarchical optimal load management scheme for aircraft power systems based on an efficient mixed integer-linear programming formulation.

6.4 Library-Based Contract Refinement Checking for Efficient Verification and Mapping

Given a global specification contract and a system described by a composition of "local" 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. When contracts are captured using high-level declarative languages, such as temporal logic, refinement checking reduces to a temporal logic satisfiability checking problem, which can be very expensive to solve for large composite contracts. When performing optimized mapping, as mentioned in Section 6.1.2, such a verification tasks may be repeated several times, which further exacerbates the problem.

In this and the next sections, we propose a *scalable refinement checking* approach that relies on the library of contracts and a set of "local" refinement assertions that enrich the library. 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 first provide some more background material and the mathematical formulation for the contract refinement checking problem in Section 6.4.1 and Section 6.4.2, respectively. Then, in Section 6.5 we illustrate our algorithm and its benefits on the aircraft electric power system case study, with up to two orders of magnitude improvement in terms of execution time with respect to conventional approaches.

6.4.1 More Background on Contract Refinement Checking

Refinement checking is arguably a key task for the successful deployment of a contract-based methodology. 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 a composite contract refines the specification contract. Even if refinement checking can be carried out compositionally, it can still be very expensive to solve for large composite contracts. For instance, when contracts can be captured using LTL, refinement checking reduces to an LTL satisfiability checking problem, which is PSPACE-complete [189]. 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 [70], 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.

To address this scalability issue within our methodology, we take inspiration by the strong growth of library-based design approaches in VLSI where, according to a recent market survey, more than 50% of components at the macro-level come from pre-designed Intellectual Property blocks (IP) that are fully characterized, pre-verified, and fully documented. The cost of building a library of IPs is non trivial but is highly compensated by the saving in design time and cost. International Business Strategies (IBS) estimates that design costs for a chip implemented in the latest technology will exceed 200 million US$ if IP libraries are not used to the fullest extent. Indeed, the market for IP blocks is now above 2 billion US$. Motivated by this trend in semiconductor companies, system companies share a growing interest in design re-use both for hardware and software. To make it possible to utilize pre-designed blocks with confidence, providing strong collateral documentation and models is necessary. Along these lines, we show how refinement checking can be made more efficient when a system is described by contracts out of a pre-characterized library that carries as collateral a characterization in terms of a set of refinement assertions.

While we instantiate and demonstrate it by using LTL A/G contracts, our algorithm is not bound to any specific contract framework. As in traditional assume-guarantee proof strategies, we decompose the main verification task into smaller sub-tasks, where an aggregation of components is replaced by a more abstract representation [89]. 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 [64, 91]. In our approach, 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, while clearly inspired by the platform-based design paradigm [176], where a design at each abstraction layer is built out of components from a pre-characterized library with composition rules, we further extend the concept of library to also include *refinement rules*.

As in previous work [58], we 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. However, in addition to automatically generate proof obligations, the contribution of our work is twofold: (i) we propose an algorithm to improve the performance of refinement checking, the core verification task underlying any proof obligation in contract-based design (CBD); (ii) we illustrate the benefits of a library-based approach for contract-based verification on a case study of industrial relevance.

6.4.2 Problem formulation

We start by recalling the definition of the refinement relation widely used in our verification algorithm. We say that contract $\mathcal{C}_1 = (V, \varphi_{e1}, \varphi_{s1})$ refines contract $\mathcal{C}_2 = (V, \varphi_{e2}, \varphi_{s2})$, both in saturated form, written $\mathcal{C}_1 \preceq \mathcal{C}_2$, if formulas $\varphi_{e2} \rightarrow \varphi_{e1}$ and $\varphi_{s1} \rightarrow \varphi_{s2}$ are both valid, or equivalently, if $\neg(\varphi_{e2} \rightarrow \varphi_{e1})$ and $\neg(\varphi_{s1} \rightarrow \varphi_{s2})$ are both unsatisfiable. Moreover, in our framework, we aim to specify systems that are built as aggregation and interconnection of components. To do so, we also define a set of operations that manipulate contract variables. A first operation on contract variables is *instantiation*. Given a set of contracts $\mathcal{C}$, defined on a universe of variables $V_{\mathcal{C}}$, an instance of a contract $\mathcal{C} = (V, \varphi_e, \varphi_s) \in \mathcal{C}$ is a contract $\mathcal{C}' = (V', \varphi'_e, \varphi'_s)$ obtained by $\mathcal{C}$ by renaming its variables so that all the variable names $v'_1, ..., v'_n$ in $\varphi'_e \vee \varphi'_s$ are unique in $V_{\mathcal{C}}$, i.e. they are not used by any other contract in $\mathcal{C}$. We will indicate the instantiation of a contract $\mathcal{C}$ as $inst(\mathcal{C})$. Given three contracts $\mathcal{C}$, $\mathcal{C}_1$ and $\mathcal{C}_2$, where $\mathcal{C}_1 = inst(\mathcal{C})$ and $\mathcal{C}_2 = inst(\mathcal{C})$, $\mathcal{C}$, $\mathcal{C}_1$ and $\mathcal{C}_2$ will not share any variable.

We then define a *renaming* operator. Given a contract $\mathcal{C} = (V, \varphi_e, \varphi_s)$, where φ_e, φ_s are defined on variables $V = \{v_1, ..., v_n\}$, then a renaming for $\mathcal{C}$ is a set M of pairs of the form (v_i, u_j), where u_j is a new variable or an existing variable v_k. The renaming operator $ren_M(\mathcal{C})$ returns a new contract $\mathcal{C}' = (V', \varphi'_e, \varphi'_s)$ where variables V' in φ_e, φ_s are renamed according to M. We will say that two contracts $\mathcal{C}_1$ and $\mathcal{C}_2$ are *isomorphic* if there exists a renaming M such that $ren_M(\mathcal{C}_1) = \mathcal{C}_2$. With the exception of their variable names, isomorphic contracts represent the same contract.

Finally, given a contract $\mathcal{C}$, we define the operations $input(\mathcal{C})$ and $output(\mathcal{C})$, which return, respectively, the list of input and output variables of $\mathcal{C}$.

6.4.2.1 The Refinement Check Problem (RCP)

Given a set of contracts $\mathcal{C}$, a composition of contracts specifying a system $\mathcal{C}_s = ren_M(inst(\mathcal{C}_1) \otimes inst(\mathcal{C}_2) \otimes \cdots \otimes inst(\mathcal{C}_n))$, where $\mathcal{C}_1, \ldots, \mathcal{C}_n \in \mathcal{C}$ and M is a renaming, and a property expressed as a contract $\mathcal{C}_p$, to ensure that any implementation of $\mathcal{C}_s$ satisfies $\mathcal{C}_p$ and can operate in all environments admitted by $\mathcal{C}_p$, we need to verify that $\mathcal{C}_s \preceq \mathcal{C}_p$. We denote this verification task as the *refinement check problem* (RCP).

For LTL A/G contracts, 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.

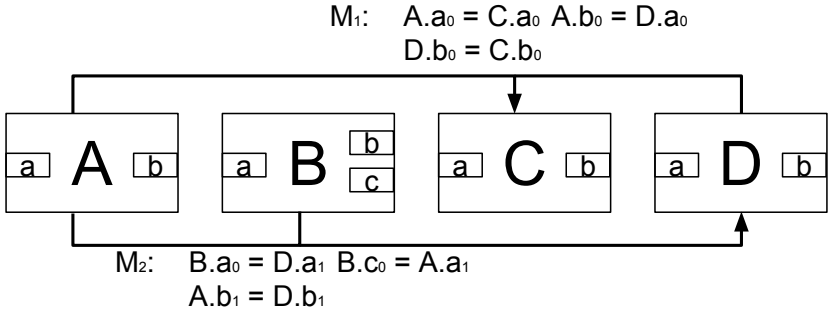


Figure 6.3: Example contract library with refinement assertions.

To perform such task more efficiently, we recur to a different problem formulation, which relies on a *library of contracts* as an additional input.

6.4.2.2 Library of Contracts and Library Verification Problem (LVP)

To describe our algorithm, in this chapter, we provide a formalization of a library of contracts $\mathcal{L}$ as a pair $(\mathcal{C}, \mathcal{R})$ where:

- $\mathcal{C} = \{\mathcal{C}_1, ..., \mathcal{C}_n\}$ is a finite set of contracts.
- $\mathcal{R}$ is a finite set of *refinement relations* between contracts in $\mathcal{C}$. Every refinement relation has the form $\mathcal{R}_i = (\mathcal{C}_{Ri}, \mathcal{C}_{Ai}, M_i)$, where $\mathcal{C}_{Ri} = ren_{M_i}(inst(\mathcal{C}_{i_1}) \otimes \cdots \otimes inst(\mathcal{C}_{i_k}))$ and $\mathcal{C}_{Ri} \preceq \mathcal{C}_{Ai}$ for $k > 1$, $\mathcal{C}_{i_1}, ..., \mathcal{C}_{i_k}, \mathcal{C}_{Ai} \in \mathcal{C}$, and M_i is a renaming for contract $inst(\mathcal{C}_{i_1}) \otimes \cdots \otimes inst(\mathcal{C}_{i_k})$. If $k = 1$, we require $\mathcal{C}_{Ri} \prec \mathcal{C}_{Ai}$, that is, $\mathcal{C}_{Ri}$ strictly refines $\mathcal{C}_{Ai}$, meaning that the two contracts cannot have equivalent formulas. This constraint is introduced to avoid, in the library, the presence of circular dependencies, and therefore ensure termination of the algorithms presented below. We will call the contract $\mathcal{C}_{i_1}$ the *root* of $\mathcal{R}_i$.

Refinement relations are *assertions* made by library designers based on their knowledge of the system architecture at hand. We say that the library $\mathcal{L}$ is *valid* if all its refinement relations are true. The *library verification problem* (LVP) is the problem of checking whether a given library is valid.

Figure 6.3 shows an example of a contract library and its refinement relations. In this case, $\mathcal{L}_{ex} = (\mathcal{C}_{ex}, \mathcal{R}_{ex})$ where $\mathcal{C}_{ex} = \{\mathcal{A}, \mathcal{B}, \mathcal{C}, \mathcal{D}\}$, and $\mathcal{R}_{ex} = \{\mathcal{R}_1, \mathcal{R}_2\}$ with $\mathcal{R}_1 = (ren_{M_1}(inst(\mathcal{A}) \otimes inst(\mathcal{D})), \mathcal{C}, M_1)$ and $\mathcal{R}_2 = (ren_{M_2}(inst(\mathcal{A}) \otimes inst(\mathcal{B})), \mathcal{D}, M_2)$.

6.4.2.3 The Refinement Check Problem with Library (RCPL)

When a library of contracts defined as in Section 6.4.2.2 is available as an additional input, a *refinement check problem with library* (RCPL) can be formulated as follows. Given a prop-

erty contract $\mathcal{C}_p$, a contract library $\mathcal{L} = (\mathcal{C}, \mathcal{R})$ and a system contract $\mathcal{C}_s = ren_M(inst(\mathcal{C}_1) \otimes inst(\mathcal{C}_2) \otimes \cdots \otimes inst(\mathcal{C}_n))$, for a renaming M, and such that $\mathcal{C}_1, \mathcal{C}_2, \ldots, \mathcal{C}_n \in \mathcal{C}$, check whether $\mathcal{C}_s \preceq \mathcal{C}_p$.

6.5 Scalable Contract Refinement Checking Algorithm

6.5.1 Library Verification

Given a library defined as in Section 6.4.2.2, 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.

Algorithm 1: *Library Verification Problem.*

Input: A library of contracts, $\mathcal{L} = (\mathcal{C}, \mathcal{R})$.
Output: `true`, if all refinement relations in the library are true, `false` otherwise.

1. For each tuple $(\mathcal{C}_{Ri}, \mathcal{C}_{Ai}, M_i) \in \mathcal{R}$, $\mathcal{C}_{Ri} = ren_{M_i}(inst(\mathcal{C}_{i_1}) \otimes \cdots \otimes inst(\mathcal{C}_{i_k}))$, $k \geq 1$, $\mathcal{C}_{i_1}, ..., \mathcal{C}_{i_k}, \mathcal{C}_{Ai} \in \mathcal{C}$, and M_i a renaming
   a) if $k > 1$ and $\mathcal{C}_{Ri} \not\preceq \mathcal{C}_{Ai}$ then return an error.
   b) if $k = 1$ and $\mathcal{C}_{Ri} \not\prec \mathcal{C}_{Ai}$ then return an error.
2. If no errors are found, then return `true`.

Each refinement check in Algorithm 1 is performed by solving an RCP instance as described in Section 6.4.2.1, 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 library verification routine is deemed to be less critical since it is performed only once, outside of the main verification flow.

6.5.2 Refinement Check with Library

Our refinement checking procedure is described in Algorithms 2, 3 and 4. We start with a valid library $\mathcal{L} = (\mathcal{C}, \mathcal{R})$, defined as in Section 6.4.2.2, a property contract $\mathcal{C}_p$ (where possibly $\mathcal{C}_p \notin \mathcal{C}$), and a system contract $\mathcal{C}_s$, obtained as the composition of a set of contracts $S = \{\mathcal{C}_1, \ldots, \mathcal{C}_n\}$, $\mathcal{C}_1, \ldots, \mathcal{C}_n \in \mathcal{C}$, after instantiation and renaming, as defined in Section 6.4.2. The system contract $\mathcal{C}_s$ represents the specification of a complex system, while the property contract $\mathcal{C}_p$ captures a requirement that must be satisfied by the system. We further assume that, given a variable v such that $v \in output(\mathcal{C}_i)$, $\mathcal{C}_i \in S$, then $v \notin output(\mathcal{C}_j)$, $\mathcal{C}_j \in S$

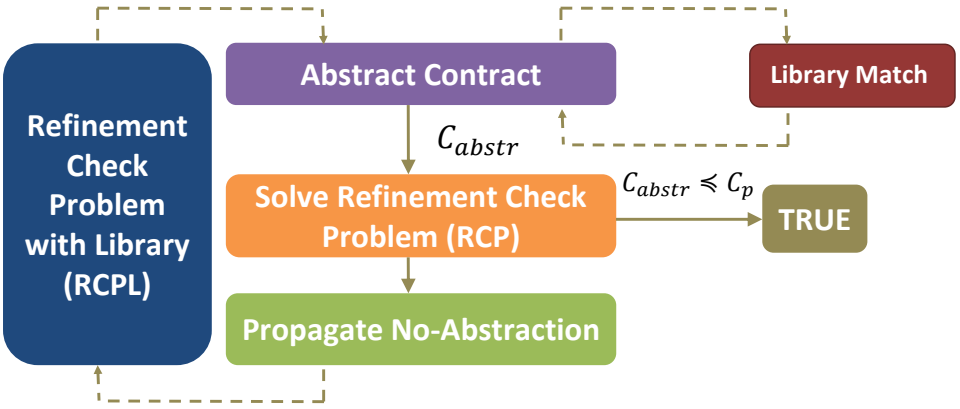


Figure 6.4: Representation of the refinement checking algorithm with library.

and $j \neq i$, meaning that each variable is controlled only by one contract in S or by a legal environment of $\mathcal{C}_s$.

We solve the RCPL using the algorithm represented in Figure 6.4 and consisting of two nested loops. In the inner loop, the procedure AbstractContract tries to create a maximal abstraction for $\mathcal{C}_s$ given the refinement assertions in $\mathcal{L}$ 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 $\mathcal{C}_{abstr}$. Since, in general, a more abstract contract is expressed by smaller formulas, $\mathcal{C}_{abstr}$ will be simpler and more compact than $\mathcal{C}_s$. The indication on which contracts can be abstracted is provided via the outer loop by the routine PropagateNoAbstraction.

In the outer loop, refinement between $\mathcal{C}_{abstr}$ and $\mathcal{C}_p$ is checked by the RCP routine. If $\mathcal{C}_{abstr} \preceq \mathcal{C}_p$ holds, then $\mathcal{C}_s \preceq \mathcal{C}_p$ will also hold since, by construction, we have $\mathcal{C}_s \preceq \mathcal{C}_{abstr}$ 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 $\mathcal{C}_s$. In the worst case, no abstraction is performed and RCPL reduces to an instance of RCP with the most concrete, "flat," contract representation. The outer loop of the RCPL procedure is illustrated in Algorithm 2. To control the level of abstraction, each contract (including $\mathcal{C}_p$) 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 6 in Algorithm 2, the main loop terminates when $\mathcal{C}_{abstr} \preceq \mathcal{C}_p$ or when the function AbstractContract cannot return a more abstract contract.

Algorithm 2: *RCPL Routine*

Input: A contract $\mathcal{C}_p$, a library of contracts $\mathcal{L} = (\mathcal{C}, \mathcal{R})$, a contract $\mathcal{C}_s$ obtained by composition of $\mathcal{C}_1, \ldots, \mathcal{C}_n \in \mathcal{C}$.
Output: `true`, if $\mathcal{C}_s \preceq \mathcal{C}_p$, `false` otherwise.

1. Let $S = [\mathcal{C}_1, \ldots, \mathcal{C}_n, \mathcal{C}_p]$.
2. Build hashtable A such that $A[\mathcal{C}_p] =$ `true` and $\forall i$, $A[\mathcal{C}_i] =$ `false`.
3. $S' :=$ AbstractContract$(A, \mathcal{L}, \mathcal{C}_s)$
4. $\mathcal{C}_{abstr} := \otimes\{\mathcal{C}_i \mid \mathcal{C}_i \in S'\}$
5. If $\mathcal{C}_{abstr} \preceq \mathcal{C}_p$ return `true`.
6. Create a copy $\mathcal{C}_{s'} = \mathcal{C}_s$.
7. While $\mathcal{C}_{abstr} \not\preceq \mathcal{C}_p$ and $\mathcal{C}_{abstr} \neq \mathcal{C}_{s'}$
   a) $\mathcal{C}_{s'} := \mathcal{C}_{abstr}$,
   b) $A :=$ PropagateNoAbstraction(A, S),
   c) S':=AbstractContract$(A, \mathcal{L}, \mathcal{C}_s)$
   d) $\mathcal{C}_{abstr} := \otimes\{\mathcal{C}_i \mid \mathcal{C}_i \in S'\}$
8. If $\mathcal{C}_{abstr} \preceq \mathcal{C}_p$ return `true`, otherwise `false`

Algorithm 3: AbstractContract

Input: A library of contracts $\mathcal{L} = (\mathcal{C}, \mathcal{R})$, a composite contract $\mathcal{C}_s$ (obtained by composition of $\mathcal{C}_1, \ldots, \mathcal{C}_n \in \mathcal{C}$), a hashtable A as in Algorithm 2.
Output: A list of contracts $S = [\mathcal{C}_{a_1}, \ldots, \mathcal{C}_{a_m}]$, such that $\mathcal{C}_s \preceq \mathcal{C}_{a_1} \otimes \cdots \otimes \mathcal{C}_{a_m}$.

1. Create $S := [\mathcal{C}_1, \ldots, \mathcal{C}_n]$.
2. Create $\mathcal{C}_{abstr} :=$ `null`.
3. Create a copy $\mathcal{C}_{s'} := \mathcal{C}_s$.
4. While $\mathcal{C}_{abstr} \neq \mathcal{C}_{s'}$
   a) assign $\mathcal{C}_{s'} = \mathcal{C}_{abstr}$,
   b) create copy $S' := S$;
   c) for each contract $\mathcal{C}_k \in S \cap S'$ that satisfies *abstraction-acceptance-condition* then
      i. $S' := (S' - [\mathcal{C}_k, \mathcal{C}_{k_1}, \ldots, \mathcal{C}_{k_m}]) \cdot [ren_N(\mathcal{C}_{Ai})]$;
      ii. $\mathcal{C}_{abstr} := \otimes\{\mathcal{C}_i \mid \mathcal{C}_i \in S'\}$.
   d) $S := S'$;
5. return S

The procedure AbstractContract in Algorithm 3 implements the inner loop of RCPL. It accepts as inputs a library $\mathcal{L} = (\mathcal{C}, \mathcal{R})$, a contract $\mathcal{C}_s$ composed of contracts in $\mathcal{C}$, and a list of flags A built as described in Algorithm 2. The algorithm tries to abstract $\mathcal{C}_s$ by using the information in $\mathcal{L}$ until no progress is made. Line 4 describes how the abstraction is performed in terms of the operations defined in Section 6.4.2. At each iteration, a copy of the current list of contracts S is maintained in S' and the *abstraction-acceptance-condition* is checked on each contract $\mathcal{C}_k \in S \cap S'$. If it evaluates to true, a subset of contracts is matched to an aggregation of contracts $\mathcal{C}_{Ri}$ in $\mathcal{L}$ and then replaced by its abstraction $\mathcal{C}_{Ai}$. The *abstraction-acceptance-condition* requires the following sub-conditions to hold:

- $\mathcal{C}_k$ is not flagged by a *no-abstraction* constraint, that is $A[\mathcal{C}_k] = \texttt{false}$;
- $\exists R_i = (\mathcal{C}_{Ri}, \mathcal{C}_{Ai}, M_i) \in \mathcal{R}$ such that $\mathcal{C}_k$ and the root of R_i are isomorphic;
- $\exists \mathcal{C}_{k_1} \dots \mathcal{C}_{k_m} \in S \cap S'$, such that $A[\mathcal{C}_{k_1}] = \dots = A[\mathcal{C}_{k_m}] = \texttt{false}$, and a renaming N such that $ren_N(\mathcal{C}_{Ri}) = \mathcal{C}_k \otimes \mathcal{C}_{k_1} \otimes \dots \otimes \mathcal{C}_{k_m}$, i.e. there exists a subset of contract that can be abstracted and such that, when composed with $\mathcal{C}_k$, generate a contract that is isomorphic with $\mathcal{C}_{Ri}$;
- $(output(ren_N(\mathcal{C}_{Ri})) \setminus output(ren_N(\mathcal{C}_{Ai}))) \cap input(\mathcal{C}_r) = \emptyset,\ \ \mathcal{C}_r \in S' \setminus \{\mathcal{C}_k, \mathcal{C}_{k_1}, \dots, \mathcal{C}_{k_m}\}$, i.e. no substitution is made if there is some other contract in $\mathcal{C}_s$, which is not in $\{\mathcal{C}_k, \mathcal{C}_{k_1}, \dots, \mathcal{C}_{k_m}\}$, and such that at least one of its input variables is missing 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 will be considered in future implementations.

Termination of AbstractContract is guaranteed since contract $\mathcal{C}_{abstr}$ will not change after a certain number of iterations. In fact, the number of matchings of contracts in $\mathcal{R}$ performed in line 4.c) 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 $\mathcal{R}_i = (\mathcal{C}_{Ri}, \mathcal{C}_{Ai}, M_i)$, with $\mathcal{C}_{Ri} = ren_{M_i}(inst(\mathcal{C}_{i_1}) \otimes \dots \otimes inst(\mathcal{C}_{i_k}))$, there are two possible cases. If $k > 1$, after replacing $\mathcal{C}_{Ri}$ with $\mathcal{C}_{Ai}$, the number of contracts in S decreases. Obviously, this operation can only be performed a finite number of times. On the other hand, if $k = 1$, since we requires $\mathcal{C}_{Ri} \prec \mathcal{C}_{Ai}$, we will always have $\mathcal{C}_{Ai} \not\prec \mathcal{C}_{Ri}$. Therefore, it is impossible to find in the library a relation $\mathcal{R}'_i = (\mathcal{C}_{Ai}, \mathcal{C}_{Ri}, M_i)$, which would represent a circular dependency between $\mathcal{R}_i$ and $\mathcal{R}'_i$.

The runtime of AbstractContract 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 $\mathcal{C}_s$. Such a matching problem can be reduced to a graph isomorphism problem, which can be efficiently solved [196, 21]. 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.

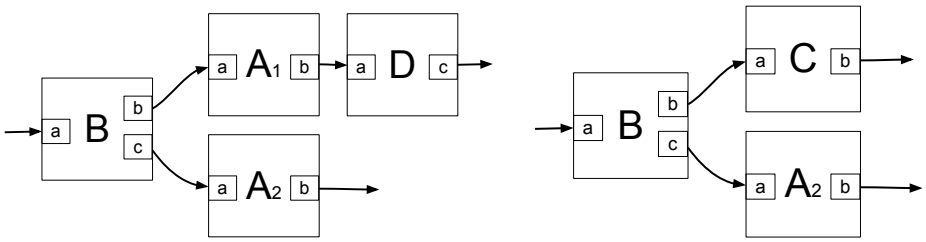


Figure 6.5: Representation of a composite contract obtained from the library in Figure 6.3 (a) and its abstraction (b).

Algorithm 4: PROPAGATENOABSTRACTION

Input: A list of contracts $S = [\mathcal{C}_1, \ldots, \mathcal{C}_n]$, a hashtable A as in Algorithm 2.
Output: A new hashtable A.

1. Create list $M := \emptyset$
2. For each contract $\mathcal{C}_k \in S$ such that $A[\mathcal{C}_k] = \texttt{true}$
   a) for each contract $\mathcal{C}_h \in S$ such that $(input(\mathcal{C}_k) \cup output(\mathcal{C}_k)) \cap output(\mathcal{C}_h) \neq \emptyset$
      i. add $\mathcal{C}_h$ to M,
3. For each contract $\mathcal{C}_i \in M$, assign $A[\mathcal{C}_i] = \texttt{true}$
4. return A.

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 $\mathcal{C}_s$, extended with the addition of the property contract $\mathcal{C}_p$ (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, for instance, in Counterexample-Guided Abstraction Refinement [62].

We provide an example of execution of our algorithm in Figure 6.5. The contract in Figure 6.5 (a) is obtained by composition of contracts from the library in Figure 6.3. The arrows denote a renaming operation. We assume that the property contract $\mathcal{C}_p$ involves only variables $\mathcal{B}.a$ and $\mathcal{D}.c$. We then call the RCPL algorithm using $\mathcal{C}_p$, $\mathcal{C}_s$ in Figure 6.5 (a), and $\mathcal{L}_{ex}$ from Section 6.4.2.2. At the first execution of ABSTRACTCONTRACT, all contracts can be potentially abstracted. However, there are only two possible matchings between portions of the architecture in Figure 6.3 and the refinement relations in $\mathcal{R}_{ex}$. In

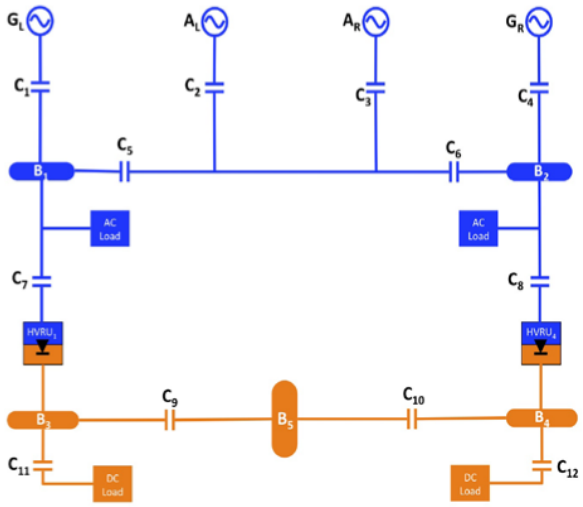


Figure 6.6: Aircraft electric power system plant architecture used to demonstrate the refinement checking algorithm with library.

particular, the composite contract $\mathcal{B} \otimes \mathcal{A}_2$ can be abstracted as $\mathcal{D}_1$, an instance of $\mathcal{D}$, while $\mathcal{A}_1 \otimes \mathcal{D}$ can be abstracted as $\mathcal{C}$. However, $\mathcal{B} \otimes \mathcal{A}_2$ does not satisfy the last condition for the *abstraction-acceptance-condition* to hold in line 4.c) of Algorithm 3. In fact, replacing $\mathcal{B} \otimes \mathcal{A}_2$ with $\mathcal{D}_1$ would cause the loss of a variable ($\mathcal{B}.b$) that should be shared with $\mathcal{A}$, hence an incorrect abstraction. Conversely, the substitution of $\mathcal{A}_1 \otimes \mathcal{D}$ with $\mathcal{C}$ is legal and the resulting contract composition, $\mathcal{C}_{abstr}$, is shown in Figure 6.5 (b). If $\mathcal{C}_{abstr} \preceq \mathcal{C}_p$, the algorithm would terminate by executing an instance of the RCP on a more compact representation of the system contract. Otherwise, if $\mathcal{C}_{abstr} \not\preceq \mathcal{C}_p$, PROPAGATENOABSTRACTION would mark $\mathcal{D}$ with a *no-abstraction* annotation. At this point, no contract aggregation can be further abstracted, and the algorithm terminates by solving an instance of the RCP on the original composition.

6.5.3 Application Example

The proposed algorithm was implemented in PYTHON and applied on the verification of a controller for an aircraft electric power system (EPS). To solve the LTL satisfiability problems, we used NUSMV [59]. All tests were performed on a 2.3-GHz Intel Core i7 machine with 8 GB of RAM.

Figure 6.6 shows the architecture of the EPS plant considered in this example. The set of components includes primary generators (G_L, G_R), auxiliary generators (A_L, A_R) on both the left and right side of the diagram, contactors $(c_1, ..., c_{12})$, buses $(B_1, ..., B_5)$, high-voltage rectifier units (HVRU_i) and loads. The EPS controller must appropriately open or close the contactors (electromechanical switches) to ensure that loads are always adequately

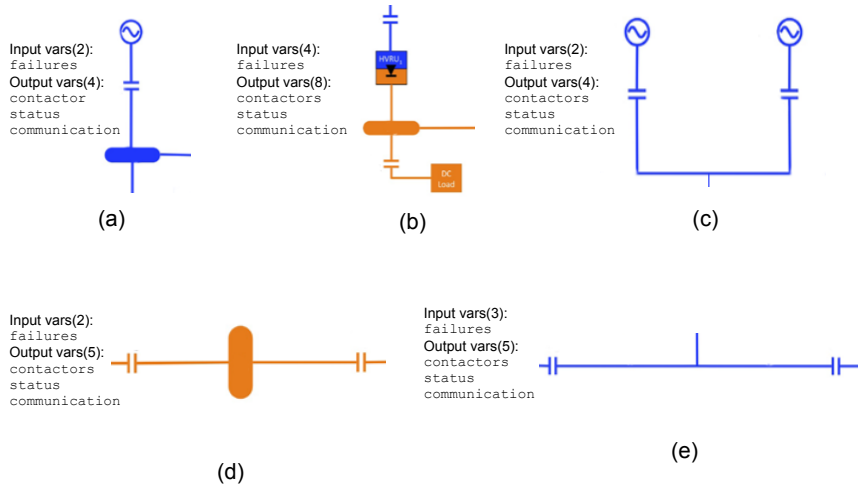


Figure 6.7: Subsets of components of the electrical power system plant and number of variables associated with the related contracts, including communication variables and variables related to the health status of plant components (e.g. buses, contactors).

powered by accommodating the highest possible number of failures in the components. In our example, we assume that failures can only affect generators and rectifiers.

Each contract in our library specifies a "local" controller for a portion of the EPS plant, i.e. a subset of its components. In addition to sensing (input) and actuation (output) variables, contracts can include a set of communication variables to propagate information on error conditions and component health status. Figure 6.7 shows some of the EPS subsystems supported by our library. A contract for the subsystem in Figure 6.7 (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 6.7 (c), the same requirement as for Figure 6.7 (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 6.7 (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 6.7 (b) specifies that the load should be isolated in case of failures in one of the interconnected portions of the plant, or in the HVRU. Finally, the subsystem in Figure 6.7 (d) is associated to a contract similar to the one in Figure 6.7 (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, which abstracts both the sub-systems in Figure 6.7 (a) and 6.7 (c). A pictorial representation of this refinement relation between local contracts is shown in Figure 6.8. Overall, the library includes 17 contracts and 9 refinement assertions. The verification of the refinement assertion using

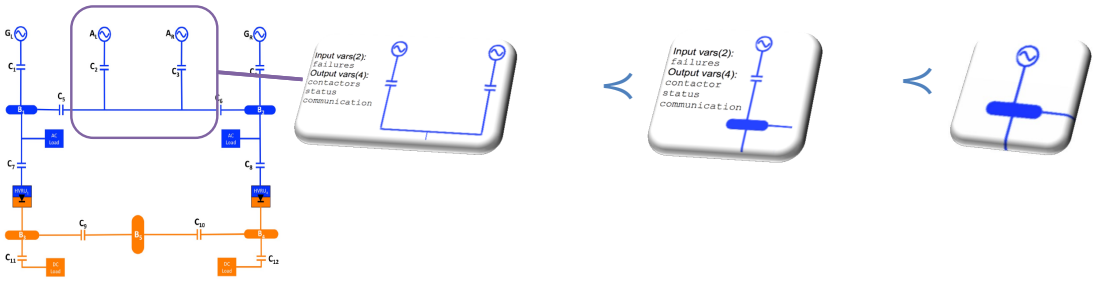


Figure 6.8: Example of refinement relations between local contracts.

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 6.7. 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:

- $\mathcal{C}_{p1}, ..., \mathcal{C}_{p4}$: 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;
- $\mathcal{C}_{p5}$: If generators G_L and G_R are healthy, contactors c_5 and c_6 must be opened;
- $\mathcal{C}_{p6}$: Contactors c_2 and c_3 cannot be both closed at the same time;
- $\mathcal{C}_{p7}, ..., \mathcal{C}_{p10}$: If at least one generator is healthy, bus $B \in \{B_1, ..., B_4\}$ cannot stay unpowered for more than three clock cycles;
- $\mathcal{C}_{p11}$: If all generators are healthy, bus B_5 must not be powered;
- $\mathcal{C}_{p12}, \mathcal{C}_{p13}$: 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 6.9 shows the execution times required by each verification task. For more than half of the properties ($\mathcal{C}_{p1}, \mathcal{C}_{p2}, \mathcal{C}_{p3}, \mathcal{C}_{p4}, \mathcal{C}_{p5}, \mathcal{C}_{p6}, \mathcal{C}_{p11}$), RCPL allows to obtain a performance improvement of two orders of magnitude, by using an abstraction of the controller with only 12 variables. For $\mathcal{C}_{p7}$ and $\mathcal{C}_{p8}$ 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. $\mathcal{C}_{p10}$ produced the worst execution time, using an abstraction with 37 variables. Figure 6.10 shows the difference in terms of formula sizes, computed as the ratio

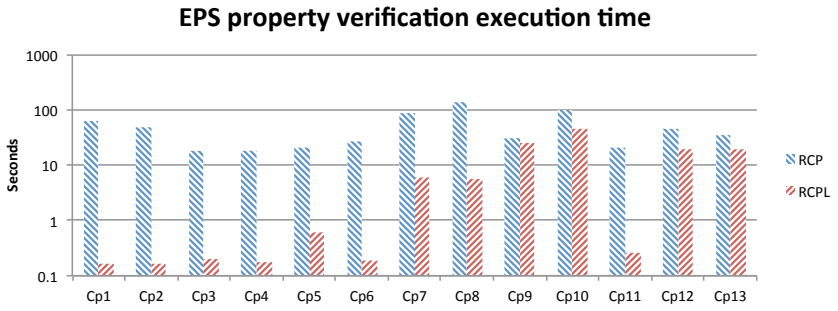


Figure 6.9: Execution time of refinement checking algorithm with library (RCPL) and refinement checking algorithm (RCP) for the verification of a set of 13 property contracts in an aircraft electric power system.

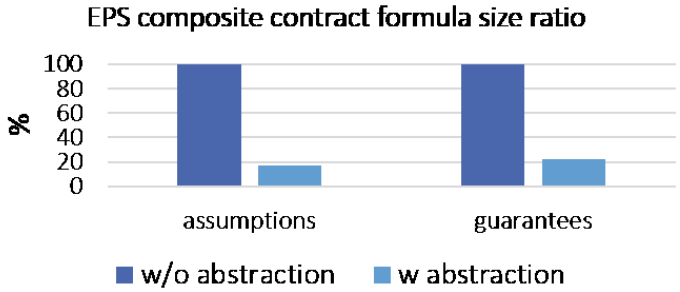


Figure 6.10: Maximum size reduction of the linear temporal logic (LTL) formulas in the abstract system contract with respect to the concrete system contract for the benchmarks in Figure 6.9.

between the non-abstract EPS contract size and the one of its maximal abstraction obtained at the first iteration of the AbstractContract 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.

6.6 Conclusions

We developed two methodologies for systematic design of control protocols within a contract-based design framework: reactive synthesis-based optimal control mapping and programming-based optimal control mapping. We showed how different vertical refinement strategies and tools (i.e., reactive synthesis, simulation-based optimized mapping, and receding horizon optimization-based control) can be effectively combined to design cyber-physical system controllers in a rigorous and scalable way. Specifically, in the context of compositional design of distributed controllers, we addressed the problem of performing scalable contract refinement checking, and presented an algorithm that leverages our pre-characterized library of contracts, enriched with refinement assertions, to break the main verification task into a set of smaller tasks. The application of the proposed algorithm to verify controllers for aircraft electrical power systems, showed up to two orders of magnitude improvement with respect to a standard implementation. Altogether, the architecture and control design paradigms introduced in this chapter and in Chapter 5 will be put into action on examples of industrial relevance in Chapter 7.

Chapter 7

Application to Aircraft System Design Examples

In this chapter we see our methodology, supporting algorithms, and tools applied to design examples of industrial relevance. We show how design exploration of architectures and control protocols for aircraft electrical power distribution can be carried out in a principled, modular way by using the Integer Linear Programming Modulo Reliability (ILP-MR) and Reactive Synthesis-Based Optimized Control Mapping (RS-OCM) methods, respectively introduced in Chapter 5 and Chapter 6. We then present a novel application of the Programming-Based Optimized Control Mapping (P-OCM) method, also introduced in Chapter 6, to the problem of power system load management with optimal resource usage, showing how load management in aircraft can indeed be made "smarter" than in current design practices. We finally provide an overview on the application of the proposed methodology to the design of an aircraft environmental control system and draw some conclusions.

7.1 Aircraft Electric Power System Design: Primary Distribution

We start by illustrating the application of the methodology introduced in this thesis to the design of supervisory control systems for aircraft power distribution. Specifically, we focus on the proof-of-concept design of the primary power distribution of an electric power system (EPS), involving the configuration of the contactors to deliver power to high-voltage AC and DC buses and loads.

A sample structure of an aircraft electric power system in the form of a single-line diagram was introduced in Figure 1.3. We briefly recall here the main components and basic system operation. *Generators* (e.g., two on the left and two on the right side of the aircraft, denoted as GEN in Figure 1.3) deliver power to the loads (e.g. avionics, lighting, heating and motors, not represented in Figure 1.3) via high-voltage and low-voltage AC (HVAC, LVAC) and DC *buses* (HVDC, LVDC). *Auxiliary* Power Unit (APU) generators or *batteries*

(Batt) are instead used when one of the primary generators fails. *Essential buses* (ESS in Figure 1.3) supply loads that cannot be unpowered for more than a predefined period t_{max}, while non-essential buses supply loads that may be shed in the case of a fault. *Contactors* are electromechanical switches that are opened or closed to determine the power flow from sources to loads, and are shown as double bars in the figure. AC *transformers* (ACT) convert high-voltage to low-voltage AC power. *Rectifier* Units (RUs) convert and route AC power to DC buses. Transformer Rectifier Units (TRUs) act both as transformers and rectifiers. The goal of the *supervisory controller* (not represented in Figure 1.3) is to react to changes in system conditions or failures and reroute power by appropriately actuating the contactors, to ensure that essential buses are adequately powered.

A pictorial overview of the proposed design flow as instantiated for the EPS was already introduced in Figure 1.6. In Chapter 5 (Section 5.7) and Chapter 6 (Section 6.2) we distilled the two main steps of the flow, i.e., architecture design and control design, and presented their application to power system design examples. In the following, we *combine* the results of Chapter 5 and Chapter 6 to show an example of application of the *overall design flow* in Figure 1.6, mapping the top-level system requirements into a lower level representation of both the plant architecture and the control algorithm, which can further be refined during the subsequent design stages. Both the architecture and control mapping phases are shown in the context of platform-based design in Figure 7.1. However, before digging into the case study, we first recall some related work on aircraft EPS design methodologies.

7.1.1 Related Work

A number of results have opened the way for a more structured approach to the design of aircraft electric power systems, by advocating the adoption of model-based development and simulation for the analysis of aircraft performance and power optimization [114, 27]. Previous work presented a platform for modeling and architectural exploration of aircraft subsystems by simulations [27]. Similarly, Krus and Nyman [114] leverage simulation using a flight dynamics model of the aircraft coupled to a model of the actuation system. In the context of the More Open Electrical Technologies (MOET) project [107], a set of model libraries were developed using the Modelica language [2] to support "more-electric" aircraft simulation, design and validation. Simulation is used for electric power system performance verification (e.g., stability and power quality) at the network level, by leveraging models with different levels of complexity to analyze different system properties, and validated with real equipment measurements. However, design space exploration, optimization and analysis of faulty behaviors in these models can still become computationally unaffordable unless proper levels of abstraction are devised, based on the goals at each design step.

A library-based approach to instantiate, analyze and verify a system design was also adopted within the META research program [116, 198], with the aim to compress the product development and deployment timeline of defense systems. A simulation framework based on Modelica was developed to enable exploration of architectural design decisions, while a language based on SysML [3] was proposed to enable semantically robust integration of

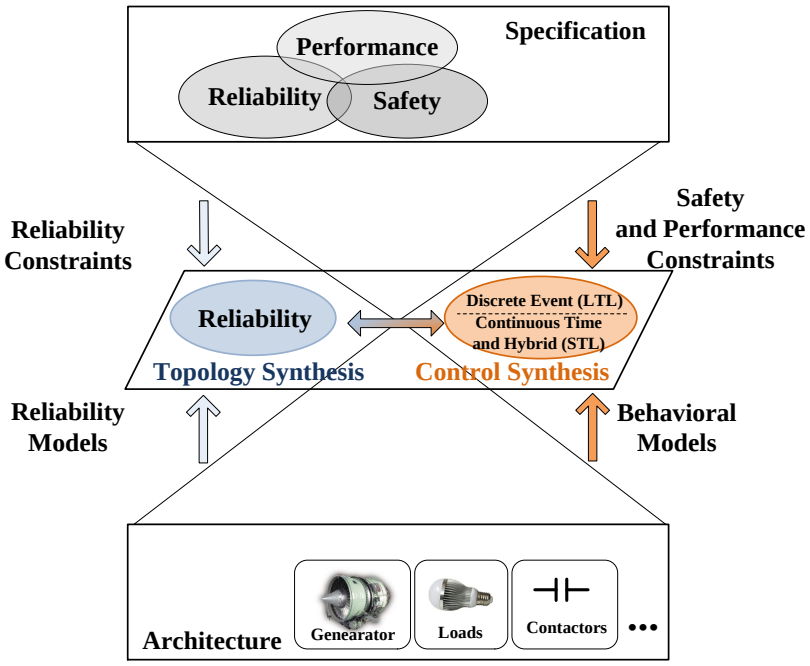


Figure 7.1: Representation of the main mapping phases in the electric power system design flow, e.g. architecture and control mapping.

models, analytical methods and results provided by other domain specific languages and tools [141]. Such integration language incorporates assume-guarantee contracts to formalize system requirements and enable the generation of monitors. In this dissertation, we further extend the use of assume-guarantee contracts as a design aid in combination with platform-based design to yield system synthesis and optimization in addition to system simulation and verification.

An optimization-oriented power system design methodology following the platform-based paradigm was also proposed, where initial specifications are refined and mapped to the final implementation in four steps [158]. At each step, a binary optimization problem is formulated to derive a class of candidate implementations for the next exploration step. The methodology deals with how to select the power generators and synthesize the electric power system topology. In this chapter, we extend the above flow [158] to enable synthesis of electric power system topology and control, subject to heterogeneous sets of system requirements that are not always approximated by binary or mixed integer-linear constraints. In particular, we build a rich, multi-view library $\mathcal{L}$ of component and contract models that can be used by different, domain-specific analysis, synthesis and verification frameworks, as described in Section 4.3. We first synthesize an electric power system topology from system requirements formalized as arithmetic constraints on Boolean variables. For the given topol-

ogy, we translate the requirements into temporal logic formulas, by which we synthesize and verify control protocols.

7.1.2 Top-Level Requirement Formalization

As a first step, top-level requirements are captured in terms of a system contract $\mathcal{C}_S$ using an electric power system domain-specific language (DSL), which enables automatic translation of the specifications from a set of pre-defined primitives to one (or more) of the back-end formalisms mentioned in Section 4.2. The proposed DSL can smoothly interface with pre-existing tools, such as visual programs for single-line diagrams, typically used by system engineers.

A first set of DSL constructs (or patterns) are designed to support the expression of structural and steady-state requirements, and their translation into mixed integer-linear constraints, as the ones used in the problem formulation of Section 5.3 and in the encoding of the approximate reliability algebra in Section 5.5. The resulting problem formulation can be given as an input to ArchEx to generate a graph $\mathcal{T}^* = (V, E)$ which represents the selected, optimal EPS architecture. Alternatively, $\mathcal{T}^*$ can be generated from a visual representation of the topology, directly drawn by the user. The set of nodes V in $\mathcal{T}^*$ represents the set of components, consisting of generators, buses, and rectifier units; the set of edges E represents the set of contactors as well as solid wire links between components. The adjacency matrix $\mathbf{e}$ of $\mathcal{T}^*$ is a square matrix whose diagonal entries are zeros, and whose non-diagonal entries are ones or zeros depending on whether a connection (with or without contactors) exists between vertices.

Based on $\mathcal{T}^*$, and a set of component properties, directly referenced from the EPS library $\mathcal{L}$, the linear temporal logic (LTL) specifications described in Sections 6.2.1.2 and 6.2.1.3 can then be automatically generated out of a second set of pre-defined DSL patterns, supporting the specification of system dynamics. Representative examples of system assumptions (A) and guarantees (G) are provided below along with examples of DSL patterns that can capture these assertions and facilitate their translation into mixed integer-linear constraints or LTL formulas.

***A*$_1$. Reliability Level.** A typical power system specification would require that the failure probability for an essential bus (i.e., the probability of being unpowered for longer than t_{max} by any of the available generators) be smaller than a certain target r_S, e.g. corresponding to 10^{-9} per flight hour. We denote the probability r_S as the *reliability level* of the system. To allow formalizing this requirement, a set of environment assumptions characterize the number and kind of component failures allowed, assuming that component failure events are all independent. For instance, in the context of control design, it is possible to automatically generate the assumptions in Section 6.2.1.2 using an "environment" primitive, for which the first input is the system reliability level, followed by all subsets of components that are uncontrolled and can fail. As an example, when only generators and rectifier units are

assumed to fail, this can be written as $\texttt{env}(r_S, \mathcal{G}_e, \mathcal{R}_e)$, where $\mathcal{G}_e \subseteq \mathcal{G}$ and $\mathcal{R}_e \subseteq \mathcal{R}$, $\mathcal{G}$ and $\mathcal{R}$ being the sets of all generators and rectifier units, respectively.

A_2. Irreversible failures. As a second set of environment assumptions, we require that when any component fails during the flight, it will not come back online. These assumptions can also be generated by an environment primitive as the one above. In the context of control design, such a primitive generates the LTL formulas in (6.13).

G_1: Reliability Level. As a first set of system-level guarantees, we need to ensure that the probability for an essential bus to be unpowered by any of the available generators r_T (i.e. the probability that there is no possible interconnection between the bus and any generator) is smaller than the required reliability level r_S, as defined above. We denote the probability r_T as the *topology reliability level*, since it relates to a system connectivity property, which is under the "responsibility" of the topology. Therefore, as the top-level contract $\mathcal{C}_S$ is partitioned to determine lower-level topology and control contracts, the topology reliability level r_T will be part of the topology contract $\mathcal{C}_T$.

Topology reliability requirements can be encoded in different ways. As a first approach, it is possible to hard-code the desired number of paths for a certain reliability. For instance, the pattern $\texttt{minimumConnections}(\mathcal{G}_m, \mathcal{B}_m, m)$ can be used to enforce that at least m paths are established between each component in $\mathcal{B}_m \subseteq \mathcal{B}$ and a component in $\mathcal{G}_m \subseteq \mathcal{G}$, where $\mathcal{G}$ is the set of generators and $\mathcal{B}$ is the set of buses in a template. Such a pattern will generate constraints as the one in (5.26). Similarly, the pattern $\texttt{maxAdmitFailures}(\mathcal{R}_f, \mathcal{B}_f, f)$ can be used to enforce that each bus in $\mathcal{B}_f \subseteq \mathcal{B}$ is connected to at least $f+1$ rectifiers in $\mathcal{R}_f \subseteq \mathcal{R}$, and can then tolerate the loss of f of them. An alternative and possibly more flexible way to specify the topology reliability would be, instead, to simply provide a set of values for the failure probabilities of critical loads or essential buses in the system, as done in Chapter 5. This approach will also be used in the case study in this chapter.

G_2: Unhealthy Sources. We require that the set of contactors directly connected to an unhealthy source be open to isolate it from the rest of the system. A "disconnect" primitive can take as input the union of subsets of $\mathcal{G}$ and $\mathcal{R}$. This primitive is written as $\texttt{disconnect}(\mathcal{G}_d, \mathcal{R}_d)$, where $\mathcal{G}_d \subseteq \mathcal{G}$ and $\mathcal{R}_d \subseteq \mathcal{R}$ and generates formulas as the ones in (6.19).

G_3: Operation in Nominal Conditions. Under nominal conditions (i.e., when all generators and rectifier units are healthy), primary generators and rectifiers on each side of the electric power system topology must provide power to the buses on the same side; all other paths (and auxiliary power units) stay inactive. At the architecture level, such a requirement can be imposed via a pattern that formulates power balance equations as the ones in Section 5.3. At the control level, we instead use a pattern that generates the formulas in (6.16).

G_4: No Paralleling of AC Sources. To avoid generator damage, AC sources should never be paralleled, i.e. no AC bus can be powered by multiple generators at the same time. A "non-paralleling" primitive for LTL formula generation accepts as inputs any subset of $\mathcal{G}$, and can be written as $\texttt{noparallel}(\mathcal{G}_p)$, where $\mathcal{G}_p \subseteq \mathcal{G}$, to generate formulas as in (6.17).

G_5: System Reaction Time. A DC essential bus can stay unpowered for no longer than t_{max} in case of failure. Let the set of all buses be $\mathcal{B}$. An "essential bus" primitive can input any subset of $\mathcal{B}$ such that the bus elements can be unpowered for no longer than the maximum allowable time. This primitive is written as $\texttt{essbus}(\mathcal{B}_e)$, where $\mathcal{B}_e \subseteq \mathcal{B}$. At the controller level, such a pattern may be used to generate a set of formulas as in (6.18), after defining a discretization step τ, corresponding to one time unit, and setting $T_s = \lfloor t_{max}/\tau \rfloor$. Additionally, an STL formula can also be generated as in (6.24), which can be monitored in a simulation environment using hybrid models.

The system requirements above, captured via patterns, are used to derive a top-level (application) contract $\mathcal{C}_T$ for the system architecture, in terms of arithmetic constraints on Boolean variables and failure probabilities (mixed integer-linear inequalities), and an application contract $\mathcal{C}_C$ for the control algorithm, expressed as a conjunction of LTL and STL contracts, as shown in Figure 1.6 (b). $\mathcal{C}_T$ captures the requirements for EPS architecture design detailed in Section 5.7, while $\mathcal{C}_C$ captures the requirements for EPS control design detailed in Section 6.2. Architecture and control protocol need to be consistently designed to satisfy $\mathcal{C}_S$, which can be guaranteed by showing that $\mathcal{C}_T \otimes \mathcal{C}_C$ (under their interconnection structure) is compatible and $\mathcal{C}_T \otimes \mathcal{C}_C \preceq \mathcal{C}_S$. We discuss the co-design problem for architecture and control algorithm in terms of vertical contract compatibility in Section 7.2.

7.2 Co-design of Primary Distribution System Topology and Control

As discussed in Section 3.4.2, to enforce the correctness of the refinement between different levels of abstraction of the design platform, including different viewpoints, contract consistency and compatibility should hold in both the horizontal and vertical directions. While the composite contract $\mathcal{C}_T \otimes \mathcal{C}_C$ can be effectively used to model the *horizontal* interaction between the controller and its plant, to guarantee the overall system reliability and real-time performance, we also need to prove compatibility and consistency of the *vertical* contracts between architecture and control for the *timing* and *reliability* viewpoints. Specifically, the control protocol makes several assumptions that must be discharged by the architecture, e.g. in terms of the topology reliability level, due to the available component redundancy, and the worst-case latency, due to delays in both the physical plant and the execution platform.

While the proofs of compatibility and correctness of $\mathcal{C}_T \otimes \mathcal{C}_C$ are performed manually in this section, reasoning with contracts is still instrumental to co-design of architecture and control. In particular, Propositions 7.2.1 and 7.2.2 below show that, if system-level requirements are "partitioned" according to $\mathcal{C}_T$ and $\mathcal{C}_C$, then the system can be designed in a compositional way, i.e., the architecture and control design steps summarized in Section 7.2.2 and Section 7.2.3 can be *independently* refined while guaranteeing that the assembled system satisfies $\mathcal{C}_S$.

More specifically, given a system reliability requirement r_S, Proposition 7.2.2 states that, if the power system topology is synthesized to implement the contract $\mathcal{C}_T$ with a reliability level $r_T \leq r_S$, then there exists a time T^* (a function of the synthesized topology and the contactor actuation delays) such that a centralized controller implementing the contract $\mathcal{C}_C$ for the given topology, with a reliability level r_S and $t_{max} \geq T^*$ can also be synthesized, and the resulting controlled system is guaranteed to satisfy the top-level requirements.

7.2.1 Independent Refinement of Topology and Control

Let $\mathcal{C}_S$ be the overall system contract, specifying a reliability requirement r_S. Let $\mathcal{C}_T$ be the application contract for the power system architecture, as defined in Section 5.7.1, and $\mathcal{C}_C$ the application contract for the control strategy, as defined in Section 6.2. We show that if system-level requirements captured by $\mathcal{C}_S$ are partitioned according to $\mathcal{C}_T$ and $\mathcal{C}_C$, then the system can be designed in a *compositional* way, i.e., the methodologies illustrated in in Section 5.7 and Section 6.2 can be *independently* deployed, while guaranteeing that the assembled system is correct and satisfies $\mathcal{C}_S$.

In particular, Propositions 7.2.1 and 7.2.2 below discuss conditions for the controlled system to satisfy the system-level contract $\mathcal{C}_S$ if the selected topology and control protocol satisfy their contracts $\mathcal{C}_T$ and $\mathcal{C}_C$. First, in Proposition 7.2.1, we assume that actuation delays are ignored in control synthesis. We then remove this assumption in Proposition 7.2.2.

Proposition 7.2.1. *Assume contactor delays are ignored in control synthesis, i.e., using the notation in Section 6.2.1,* $T_{o_{min}} = T_{o_{max}} = T_{c_{min}} = T_{c_{max}} = 0$ *(and therefore no contactor intent variable is introduced in* $\mathcal{C}_C$*). If the topology implements its contract* $\mathcal{C}_T$ *with a reliability level* r_T*, then a centralized control implementing its contract* $\mathcal{C}_C$ *for this topology is always realizable when a reliability level* $r_S \geq r_T$ *is used while generating the environment assumptions as in* (6.12)*. Moreover, the controlled system will satisfy the system-level requirements with a reliability level* r_S*.*

Proof. As shown in Figure 7.1 and Figure 1.6, both the topology synthesis and control synthesis steps are based on a consistent set of models and share the same labelled topology template $\mathcal{T}$. In fact, the topology generated from $\mathcal{C}_T$ will conform to $\mathcal{T}$, and the same synthesized topology is used to generate the LTL formulas for the controller contract $\mathcal{C}_C$. We prove the realizability of the controller by discussing the system-level requirements listed in Sections 6.2.1.2 and 6.2.1.3, for the controller, and Section 5.7 for the topology, as follows:

(a) *Reliability Requirements.* In both the topology and control design steps, we assume that when a component fails it will not come back online. Therefore, reliability requirements are treated as "static" requirements that hold in a failure scenario. If the topology guarantees a reliability level r_T, then there are enough components and paths from generators to critical loads such that any combination of component faults causing a system failure has a joint probability $p < r_T$. Let $\overline{\mathcal{E}}_T$ be set of all the environment configurations that correspond to these combinations of component faults, defined as in Section 6.2.1.2, and let $\mathcal{E}_T$ be its complement. Then any combination of faults associated with a configuration in $\mathcal{E}_T$ does not cause any loss in system functionality because of the available redundancy. Since $r_S \geq r_T$ is used in $\mathcal{C}_C$, $\mathcal{E}_S \subseteq \mathcal{E}_T$ will also hold, hence accommodating any combination of faults associated with an environment configuration in $\mathcal{E}_S$ will also be feasible. Therefore, a centralized controller assuming a reliability level r_S in $\mathcal{C}_C$ will always realize this specification, thus guaranteeing an overall reliability level $r = r_S$ for the controlled system.

(b) *Balanced Power Flow in Nominal Conditions.* Power requirements are treated as static requirements in nominal condition. Power flow constraints in the topology optimization problem enforce that loads on each side of the topology graph are selectively connected to one or more generators on the same side, in such a way that the total power capability of the generators is equal or larger than the required power from the respective loads. It is, therefore, enough to use the available paths in the synthesized topology for a centralized controller to realize this specification.

(c) *Unhealthy Sources.* Connectivity constraints in the topology optimization problem enforce that any edge (interconnection) originating from a source node (generator or rectifier unit) is associated with a contactor. Therefore, it is always possible for a centralized controller to open such contactors to isolate unhealthy sources and realize this specification. Since contactors can be instantaneously operated, full isolation of unhealthy sources is guaranteed within one time step (δ time).

(d) *No Paralleling of AC Sources.* As discussed above, all AC sources can be isolated by opening the related contactors. Moreover, connectivity constraints prescribe that AC buses be also connected via contactors. This makes it possible for a centralized controller to always realize this specification by isolating buses connected to different AC sources as well as isolating unhealthy sources while inserting healthy ones.

(e) *Safety-Criticality of Buses.* Since all contactors are assumed as ideal and instantaneously controllable, it is always possible for a centralized controller to configure the topology and realize this specification whenever $T_s \geq \delta$, δ being the discretization step assumed while synthesizing the controller (as in the formulas in Section 6.2.1).

We then conclude that the conjunction of the LTL formulas used in $\mathcal{C}_C$ to formalize requirements (b)-(e) under the assumptions in (a) can always be realized by a centralized controller if $r_T \leq r_S$ holds. □

Based on Proposition 7.2.1, for the controlled system to satisfy a contract $\mathcal{C}_S$ with a reliability level r_S, it is enough to select a topology that implements its contract $\mathcal{C}_T$ with a reliability level $r_T \leq r_S$, and then synthesize a centralized controller for the selected topol-

ogy by using a reliability level r_S to generate the environment assumptions. When contactor delays are not ignored in control synthesis, a similar proposition holds if an additional condition is assumed on the maximum bus unpowered time T_s allowed in (6.18), as discussed below.

Proposition 7.2.2. *Assume delays in the contactors are taken into account in control synthesis (i.e., $T_{o_{min}} > 0$ and $T_{c_{min}} > 0$). If the topology implements its contract $\mathcal{C}_T$ with a reliability level r_T, then there exists a large enough time T^* such that a centralized control implementing its contract $\mathcal{C}_C$ for this topology is realizable when a reliability level $r_S \geq r_T$ in equation* (6.12) *and a bus unpowered time $T_s \geq T^*$ in equation* (6.18) *are used while generating $\mathcal{C}_C$. Moreover, the controlled system will satisfy the system-level requirements with a reliability level r_S.*

Proof. As in Proposition 7.2.1, both the topology and control synthesis steps are based on a consistent set of models and share the same template $\mathcal{T}$. Moreover, we can prove the realizability of the controller by discussing the requirements in Section 6.2.1.3 and Section 5.7 one at a time. In particular, the static specifications in (a) and (b) will be always realizable by the same arguments used in Proposition 7.2.1. To show that the dynamic requirements in (c), (d) and (e) will also be realizable when actuation delays are taken into account, we proceed as follows:

(c)-(d) *Unhealthy Source Isolation and AC Sources Paralleling.* By the same arguments used in Proposition 7.2.1 (c), all sources (including AC sources) can be isolated by opening the related contactors. Moreover, the topology connectivity constraints prescribe that AC buses should also be connected via contactors. Even if contactors can only be opened or closed with a delay, it is still possible for a centralized controller to realize this specification by disconnecting two AC buses at least $T_{o_{max}}$ time before connecting them to different AC sources or by isolating unhealthy sources at least $T_{o_{max}}$ time before connecting the healthy ones.

(e) *Safety-Criticality of Buses.* To guarantee that safety-critical buses are unpowered for no longer than T_s, the controller needs to reconfigure the topology by opening and closing sets of contactors to deactivate existing components and paths and activate new ones. Because of the actuation delay, topology reconfigurations cannot occur instantaneously; some sets of contactors will need to be actuated in sequence to guarantee isolation of unhealthy sources and prevent paralleling of AC sources, as required in (c) and (d). Since there is a finite number of topology configurations, there will also be a finite number of possible reconfigurations $\mathcal{R}$. Consider the step i from an initial configuration A_i to a final configuration Z_i. Let n_o^i and n_c^i be the minimum number of contactor sets that must be, respectively, opened and closed in sequence in order to provide power to a critical bus during reconfiguration i. Then, the minimum (worst-case) time during which at least one critical bus stays unpowered will be

$$T_i = n_o^i \left\lceil \frac{T_{o_{max}}}{\delta} \right\rceil \delta + n_c^i \left\lceil \frac{T_{c_{max}}}{\delta} \right\rceil \delta.$$

Table 7.1: Components and attributes used for the electric power system case study.

Generators	g (kW)	Loads	l (kW)	Components	c
LG1	70	LL1	30	Generator	g/100
LG2	30	LL2	40	Bus	200
RG1	50	RL1	20	Rectifier	200
RG2	40	RL2	30	Contactor	100
APU	100				g/100

Let $T^* = \max_{i \in \mathcal{R}} T_i$; then, T^* is the minimum bus unpowered time that can always be guaranteed across all possible topology reconfigurations. Therefore, a centralized controller can always be realizable when $T_s \geq T^*$ is chosen in $\mathcal{C}_C$.

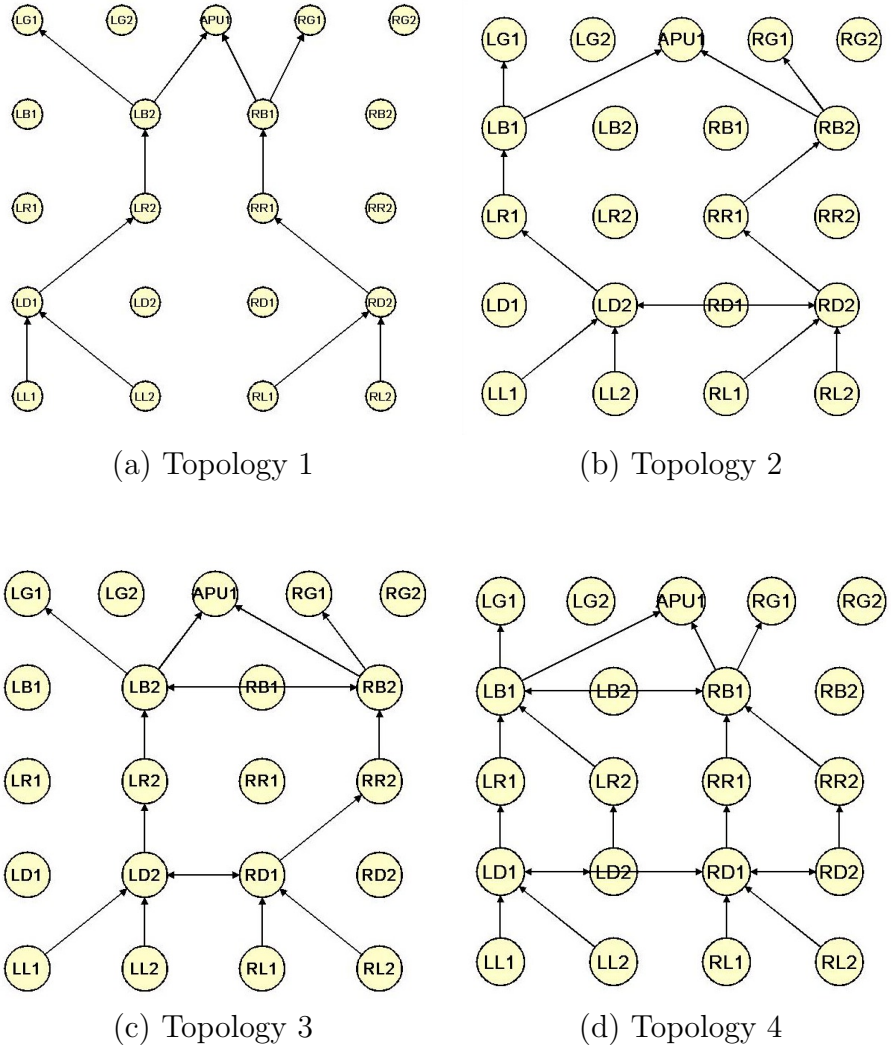
As in Proposition 7.2.1, by combining the arguments above with the ones used in (a) and (b), we can conclude that the conjunction of the LTL formulas used in $\mathcal{C}_C$ to formalize requirements (b)-(e) under the assumptions in (a) can always be realized by a centralized controller if $r_T \leq r_S$ and $T_s \geq T^*$ hold. □

7.2.2 Architecture Design

At the structural (steady-state) level of abstraction, the *plant architecture* is modelled as a directed graph, where each node represents a component (with the exception of contactors, which are associated with edges) and each edge represents an interconnection, oriented based on the direction of the power flow, as discussed in Section 5.7. The platform library $\mathcal{L}$ includes, as attributes, generator power ratings, component costs and failure probabilities, in addition to the component contracts encoding interconnection rules. The *supervisory controller* is abstracted as one or more finite state machines, which actuate the contactors in the plant to configure the network and route the power from the generators to the loads based on the failure status of the components. The controller is characterized by a reaction time T_r.

Based on the platform library described above, the contract $\mathcal{C}_T$, capturing the safety, connectivity, power flow, and reliability requirements for the system architecture can be expressed (both assumptions and guarantees) in terms of linear inequalities on a set of Boolean variables, each denoting the presence or absence of an interconnection in the topology graph, as detailed in Section 5.3 and Section 5.7. The trade-off between redundancy and cost can then be explored using ArchEx, and the synthesized topology is offered as a specification for the control refinement step. The architecture design step is shown in Figure 1.6 (c).

Table 7.1 reports the load power requirements, the generator power ratings, and the component costs in our example. We assume that generators and rectifiers fail with a probability of 10^{-5} and 2×10^{-4}, respectively, while the failure probabilities of the other components in this example are negligible. Figure 7.2 shows the topologies generated by one run of the "lazier" version of the ILP-MR method, as described in Section 5.7.2, for a



(a) Topology 1 (b) Topology 2

(c) Topology 3 (d) Topology 4

Figure 7.2: Candidate topologies for an electric power system consisting of rows of (from top to bottom) generators, AC buses, rectifier units, DC buses, and DC loads.

reliability requirement $r^* = 10^{-9}$. Instead of using LearnCons and EstPath to gather an immediate estimation of the the number of necessary paths to achieve the desired reliability, in this example, ILP-MR tries to increase the reliability by enforcing only one additional path at a time, by first adding connections among existing components, and eventually, if necessary, more components.

By solving the ILP including only connectivity and power flow constraints, we obtain the topology in Figure 7.2 (a), the simplest possible architecture, which only provides a single path from a load to a generator (or APU) on each side. Such a topology presents the highest failure probability. In Figure 7.2 b) and c) additional paths from generators to loads are added via horizontal connections between the DC buses and AC buses of the left and right

Table 7.2: Load and system failure probabilities for the topologies in Figure 7.2.

Topology	Load Failure Probability (r_T)	System Failure Probability
1	2×10^{-4}	4×10^{-4}
2	4×10^{-8}	4×10^{-8}
3	4×10^{-8}	4×10^{-8}
4	2.6×10^{-15}	2.6×10^{-15}

hand sides of the system. Additional components (e.g. buses and rectifiers) are finally used in Figure 7.2 d) to satisfy the requirement. In Table 7.2, we report the achieved reliability level r_T (failure probability) at one load as well as the overall achieved reliability level when all loads are considered, as computed for the topologies in Figure 7.2. The total computation time to generate the topologies was 19.7 s on an Intel Core i7 2.8-GHz Processor with 6-GB memory. In a typical run with LearnCons, as shown in Section 5.7.2, the number of necessary paths to achieve r^* is estimated after the first ILP instance, and convergence to the final topology occurs in no more than three iterations.

7.2.3 Control Design

To validate the RS-OCM approach, for each of the four topologies in Figure 7.2, a set of environment assumptions and controller guarantees can be generated to synthesize centralized and distributed control protocols for an overall reliability level $r_S = r_T$, as discussed in Section 7.2. As an example, we present the variables and formal specifications, written in LTL, for the topology depicted in Figure 7.2 c).

7.2.3.1 Reactive Synthesis: Centralized Controller

Environment Variables: Generators $LG1, LG2, APU1$ and rectifier units $LR2$ and $RR2$ are uncontrolled variables that can switch between healthy (1) and unhealthy (0).

Controlled Variables: Contactors $C_{i,j}$[1] (depicted only as wires in Figure 7.2) are variables that are set to open (0) or closed (1).

Dependent Variables: Buses are either powered (1) or unpowered (0) depending on the status of environment and controlled variables.

Environment Assumption: We allow environment configurations which are mapped back from the function h in Section 6.2.1.2 to an overall system reliability level r_S. Topology

[1] i and j denote the name of the components contactor $C_{i,j}$ connects.

3 from Figure 7.2 c) has a total of 32 environment configurations. For a reliability level $r_S = 4 \times 10^{-8}$, $h(r_S)$ maps to a set of 21 allowable configurations. The specification can be written as a conjunction of all configurations. More compactly, the environment assumption disallows configurations in which either both rectifiers fail or all generators fail. Thus, we can equivalently write the environment assumption for Topology 3 as:

$$\square\neg((LG1 = 0) \wedge (APU1 = 0) \wedge (RG1 = 0)) \wedge \ \square\neg((LR2 = 0) \wedge (RR2 = 0)).$$

No Paralleling of AC Sources: No combination of contactors can be closed so that a path exists between generators:

$$\square\neg((C_{LG1,LB2} = 1) \wedge (C_{APU1,LB2} = 1)) \wedge \ \square\neg((C_{APU1,RB2} = 1) \wedge (C_{RG1,RB2} = 1)).$$

Power Status of Buses: A bus can only be powered if there exists a path (in which a contactor is closed) between a bus and a generator. In Figure 7.2 c), bus $LB2$ is powered if either generator $LG1$ or $APU1$ is powered, and the contactor between generator and bus is closed:

$$\square((LG1 = 1) \wedge (C_{LG1,LB2} = 1) \rightarrow (LB2 = 1)),$$
$$\square((APU1 = 1) \wedge (C_{APU1,LB2} = 1) \rightarrow (LB2 = 1)).$$

If neither of these two cases is true, then $LB2$ will be unpowered. These specifications are written as

$$\square(\neg(((LG1 = 1) \wedge (C_{LG1,LB2} = 1) \vee ((APU1 = 1) \wedge (C_{APU1,LB2} = 1))) \rightarrow (LB2 = 0)).$$

Similar specifications may be written for buses $RB2, LD2$, and $RD1$.

Safety-Criticality of Buses: We consider all buses to be safety-critical; at the abstraction level of LTL, this is equivalent to require that at no time can any bus be unpowered

$$\square((LB2 = 1) \wedge (RB2 = 1) \wedge (LD2 = 1) \wedge (RD1 = 1)).$$

The resulting controller has 32 states with a computation time of 1.6 s on a 2.2-GHz Intel Core Processor with 4-GB memory.

7.2.3.2 Reactive Synthesis: Distributed Controller

For the topology in Figure 7.2 c), the distributed control synthesis problem can be solved by splitting the topology into two subsystems S_1 and S_2. The sets $\mathcal{E}_{S_1}, \mathcal{S}_{S_1}$, and $\mathcal{E}_{S_2}, \mathcal{S}_{S_2}$ contain all environment and system variables for subsystems S_1 and S_2, respectively. $\mathcal{E}_{S_1}$ is composed of generators $LG1$, $APU1$ and $RG1$. $\mathcal{S}_{S_1}$ contains AC buses $LB2, RB2$, and contactors $C_{LG1,LB2}, C_{APU1,LB2}, C_{RG1,RB2}$, $C_{LB2,RB2}$. $\mathcal{E}_{S_2}$ is composed of rectifiers $LR2, RR2$ and AC

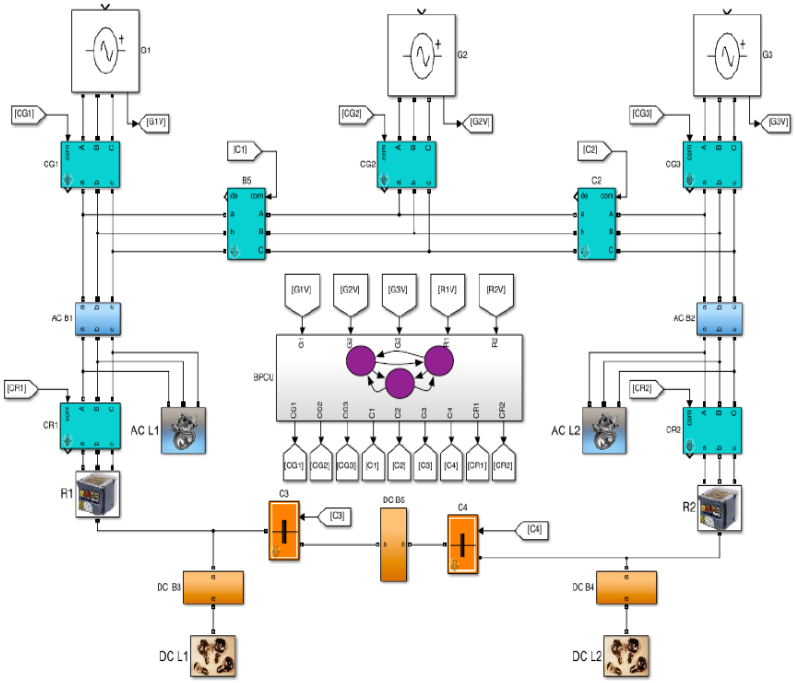


Figure 7.3: Hybrid model of the electrical power system used for simulation-based design space exploration.

buses $LB2, RB2$, while $\mathcal{S}_{S_2}$ contains DC buses $LD2, RD1$ and contactors $C_{LR2,LD2}, C_{RR2,RD1}$, $C_{LD2,RD1}$. We assume the link between AC buses and rectifier units is a solid wire.

The environment assumption $\varphi_{e_{S_1}}$ for subsystem S_1 enforces that at least one generator will always remain healthy. Environment assumption $\varphi_{e_{S_2}}$ enforces that at least one rectifier unit will always remain healthy. In addition, it also assumes that both AC buses will always be powered. This is an additional guarantee S_1 must provide to S_2 for the distributed synthesis problem to become realizable. All other specifications remain the same as the centralized control problem.

The synthesized controllers for S_1 and S_2 contains 4 and 8 states, respectively. Each controller has a computation time of approximately 0.5 s on a 2.2-GHz Intel Core Processor with 4-GB memory.

7.2.3.3 Simulation-Based Design Space Exploration

Continuous-time models of the plant are implemented in SIMULINK, by exploiting the SimPowerSystems extension, as shown in Figure 7.3. As an example, the continuous-time model of a generator consists of a mechanical engine (turbine), a three-phase synchronous machine, in addition to the generator control unit, driving the field voltage of the generator. In ad-

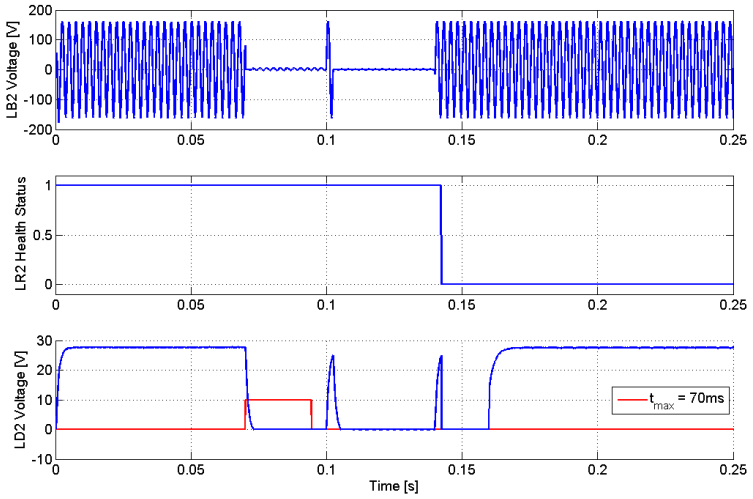


Figure 7.4: Real-time requirement violation at the DC bus $LD2$ in the topology of Figure 7.2 (c), due to a two-generator fault followed by a rectifier fault.

dition to timing properties, our power network model allows measuring current and voltage levels at the different circuit loads. It can be discretized to speed up simulations and can seamlessly interface also with MATLAB functions or STATEFLOW models implementing the controller.

In what follows, we focus on the centralized controller for Topology 3 in Figure 7.2 c), and provide results for the design exploration problem in Section 6.2.2. In particular, we are interested in finding the maximum controller reaction time T_r^* as a function of T_d, so that the essential DC bus $LD2$ is never out of range for more than $t_{max} = 70$ ms. Based on the environment assumptions discussed above, the worst case failure scenario for the left DC bus $LD2$ occurs when cascaded failures in two generators (e.g. $LG1$ and $APU1$) and one rectifier ($LR2$) correlate so as to maximize the time the bus voltage is out of the specified range. The controller reacts to a generator fault by routing power from another generator and connects the two DC buses $LD2$ and $RD1$ when one of the rectifiers fails. Therefore, the worst case failure scenario occurs when the rectifier fault happens at the end, and any fault after the first one happens right before $LD2$ fully recovers from the previous fault, while trying to reach the desired voltage level.

Figure 7.4 shows the simulated voltage V_{LD2} of bus $LD2$ as a function of time, in the worst case scenario, for $T_r = 20$ ms and $T_d = 20$ ms, both defined as in Section 6.2.2. The waveforms at the top and bottom of the figure are the voltage signals at the $LB2$ (AC) and $LD2$ (DC) buses, respectively. The signal in the middle represents the health status of $LR2$. Both the AC and DC voltages decay to zero because of the generators' faults. When a fault is also injected into $LR2$, an additional drop in the DC voltage is observed. The red signal at the bottom of the figure is interpreted as a Boolean signal, which is high (one) when χ in

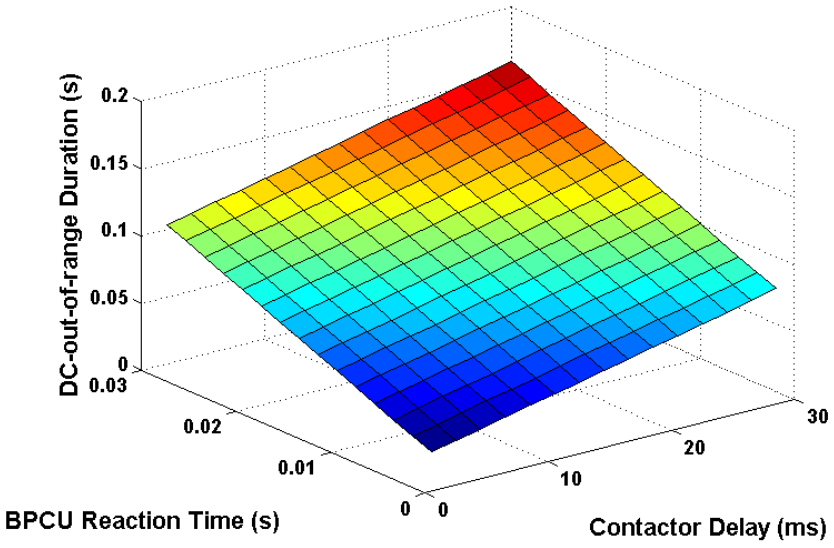


Figure 7.5: Maximum duration of the violation of the DC bus voltage requirement for the DC bus $LD2$ in the topology of Figure 7.2 (c).

equation (6.20) holds (i.e. the requirement is violated) and low (zero) otherwise. To evaluate the formula (6.20), we used $V_d = 28$ V, $\epsilon = 2$ V and $t_{max} = 70$ ms. The requirement on the DC bus is violated for 24.4 ms. Therefore, ($T_r = 20$ ms, $T_d = 20$ ms) is an unsafe parameter set.

The T_r versus T_d design space is explored in Figure 7.5 and 7.6 by following the optimization procedure in Section 6.2.2. We sampled the parameter space in approximately 4 hours to obtain a 15×15 point grid. The first plot represents the maximum amount of elapsed time τ_e^*, while the DC bus voltage is out of range, i.e. for how long the requirement on the DC bus is violated, as computed in equation (6.23). Such a violation period is then compared with the "hard" threshold $t_{max} = 70$ ms in Figure 7.6, thus providing the designer with the "safe" region (marked in blue in Figure 7.6) for the controller reaction time as a function of the contactor delay. As an example, for a specific value of $T_d = 20$ ms the maximum controller reaction time T_r^* allowed for safe operation is 6.5 ms.

7.2.3.4 Discussion

The computational complexity of monolithic synthesis from LTL contracts makes solving industrial-scale problems difficult for current tools. We therefore explored the use of distributed control protocols, which take less computational time to synthesize due to fewer components within each subsystems and, thus, smaller state spaces. A major challenge in reactive synthesis from LTL specifications is posed by timing requirements (e.g., essential bus safety and contactor open/closing times), typically addressed by nested "next" con-

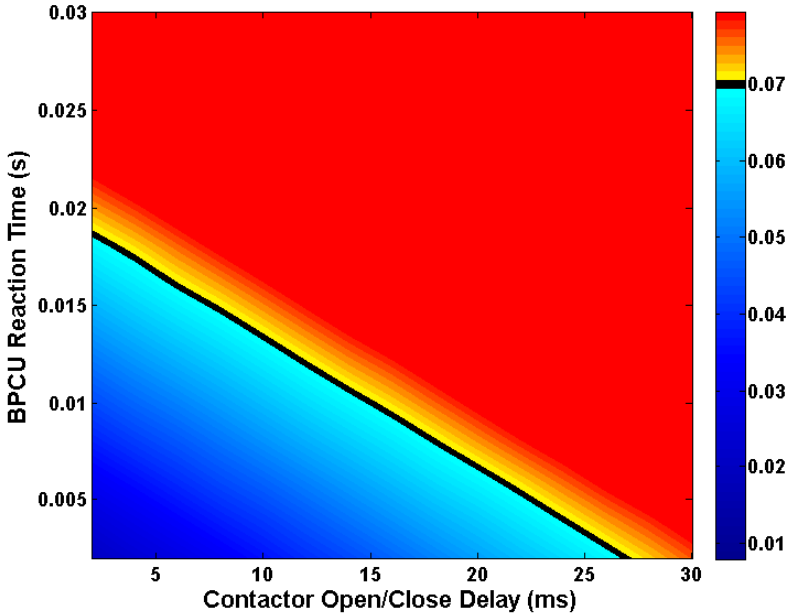


Figure 7.6: Controller (BPCU) reaction times and contactor delays in the blue region satisfy the DC bus requirement on bus $LD2$ for the topology of Figure 7.2 (c).

structs, or with the use of of additional counter variables, as detailed in Section 6.2.1.3 and Section 6.2.1.4. In our application, each contactor requires two additional counter variables to model open and closing delays, while one counter variable is needed per each essential bus, to capture the requirement on the maximum time t_{max} allowed for a bus to stay unpowered. This discretization of time further adds to the difficulties arising from the state space explosion. Usually, assuming a bounded time horizon T for the overall system operation, the discretization step τ is selected as the greatest common divider (GCD) of all the lengths of the time intervals used in the specification, while the counter variable range N is determined by the ratio between the horizon and the discretization step, i.e. $N = \lceil T/\tau \rceil$, $\lceil x \rceil$ denoting the smallest integer not less than x.

To concretely illustrate the scalability issues induced by timing requirements, we experiment on a simplified EPS architecture including only two generators, two AC buses, two rectifiers, and two DC buses, all connected through contactors as shown in Figure 7.7. We assume that the architecture includes only one essential bus, i.e. the left hand side DC bus, and use TuLiP to synthesize controllers that can accommodate a failure in the left hand side generator, by rerouting power from the right generator to the left DC bus in a time interval which is less than or equal to t_{max}. Figure 7.8 (a) and (b) show the synthesis time of a controller using the methodology in Section 7.1.2 as a function of the counter variables' range when only three and four counters are added, respectively, to handle timing requirements. In both cases, one counter is used to formalize the safety requirement on the critical bus b_6,

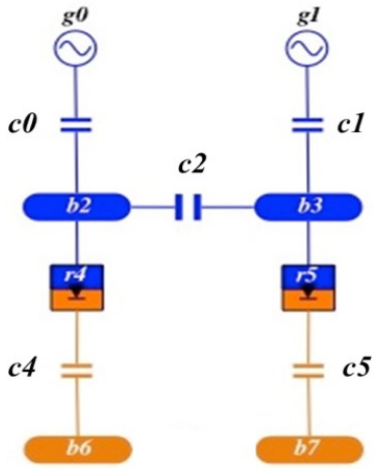
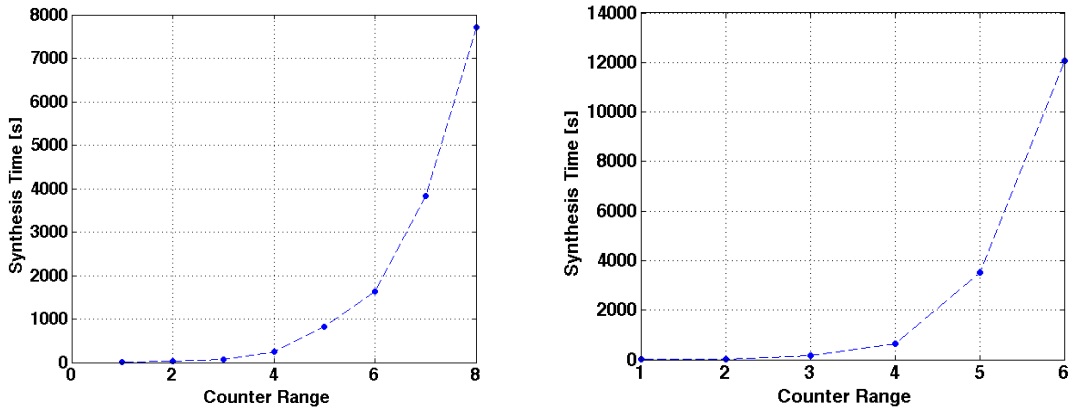


Figure 7.7: Simplified electric power system architecture used to test the scalability of reactive synthesis from linear temporal logic specifications for requirements including time intervals.



(a) (b)

Figure 7.8: Synthesis time versus counter range for a linear temporal logic specification including three counter variables (a) and four counter variables (b).

while the the others are used to model (open or closing) delays of two or three contactors, respectively, in the critical path between the generators and the essential bus. We assume that all the other contactors are "ideal," i.e. can be closed or opened within τ. We observe that the synthesis time grows exponentially with the number of counters, which is expected since the number of states of the system also grows exponentially. However, the synthesis problem becomes also expensive, even for such a simple architecture, when N increases.

In the case of four counter variables with range $N = 6$, controller synthesis takes 3.3 h, while the extrapolation of the result in Figure 7.8 (b) for a system with 16 counter variables (e.g. related to 8 contactors) and $N = 6$ would lead to a synthesis time of approximately 5 days. Noticeably, in all our experiments, checking *realizability* required, instead, much less time, always below 1 second, since it can be performed by adopting binary decision diagram (BDD) representations to reason about sets of states without enumerating all of them. Realizability checking can then be efficiently used to perform contract compatibility or consistency checking.

There is a number of directions to overcome the scalability issues above. A possible approach is to use reactive synthesis with a coarser discretization of the time intervals in the requirements and the plant behaviors when generating the LTL formulas, to reduce the number of counter variables. The timing behavior of the resulting state machine can then be fine tuned later on, by leveraging simulation and contract monitoring on a high-fidelity, hybrid model, to determine the required controller reaction time. This is the approach pursued in Section 7.2.3.3.

A second approach is to favor distributed control synthesis schemes and devise mechanisms to automate the decomposition of a global contract into a set of local contracts. An initial step in this direction could rely on counter-strategy guided refinements of the original specification to partially automate such a decomposition, using techniques similar to the ones recently proposed in the literature for generating environment assumptions [126]. Alternatively, by following a bottom-up approach, it is also possible to design distributed controllers via optimized mapping of the global contract into an aggregation of local contracts, using a contract refinement checking routine in the optimization loop, as discussed in Section 6.4.

Finally, when both the requirements and the system dynamics can be efficiently encoded into an optimization problem over a finite time horizon, we can use the Programming-Based Optimized Control Mapping method illustrated in Section 6.3 to find a control strategy, which shows better scaling properties when applied to the load management problem discussed in Section 7.3 and Section 7.4.

7.3 Aircraft Electric Power System Design: Load Management

The automated procedure for correct-by-construction design of EPS control protocols based on reactive synthesis from LTL contracts, as deployed in Section 7.2, proves to be successful in guaranteeing a set of safety, reliability, and real-time performance requirements. However, while the correctness of the final solution is guaranteed, the optimization step may occur quite late in the design flow, when several design decisions have already been taken, thus leaving a lot on the table in terms of performance optimality. For instance, the RS-OCM scheme succeeds in optimizing some of the timing properties of the controller; however, the optimality of the algorithm with respect to a number of performance metrics, such as the

number of used generators and shed loads, is not addressed. Such a limitation adds to the potential shortcomings discussed above, due to the underlying computation complexity of reactive synthesis from LTL specification, and the need to operate a possibly coarse discretization and quantization of the continuous dynamics during the initial phase of the design, which may be unacceptable for certain applications.

While several optimization techniques have been reported for the power electronics, switches, and converters in power systems, the problem of optimizing the overall power distribution system has received scant attention in the literature. Chandrasekaran et al. [54] propose to determine the optimum voltage and power levels at various points of the network to minimize the total weight, installation costs and fuel consumption. However, the main focus of this approach as well as the one previously mentioned in Section 7.1.1 [158] is on the selection of the generators and the design of the topology, i.e. the interconnection among different EPS components, rather than optimal design of the switching logic for the EPS contactors and load management.

In this section, we address the problem of designing an optimal control strategy for the EPS contactors in the presence of system faults. We offer an alternative formulation of the EPS control problem as a Mixed-Integer Linear Program (MILP) that can be efficiently solved to yield load shedding, source allocation, contactor switching and battery charging policies that are correct and optimize performance metrics, such as the number of used generators and shed loads. Since safety is of paramount importance for the application, the control scheme has to quickly react in the event of unexpected changes in loads or component failures. To do so, we propose a two-level hierarchical scheme where a high-level load management system receives as inputs the required-power prediction for each bus over a time horizon of interest, the health status (operational or faulty) of power sources and contactors, the whole set of system requirements, and solves the optimal control problem within a receding horizon approach. The output is a piece of "advice" for the low-level load management system, which handles system faults by directly actuating the EPS contactors and decides to implement such advice only if it is safe. Our goal is twofold:

- From an application standpoint, we aim to extend the capabilities of the BPCU designed in Section 7.2 by augmenting it with an optimal *load management system* based on the same formalization of the connectivity, safety and performance requirements used in Section 5.7, Section 6.2, Section 7.1.2, and Section 7.2;

- From a methodology standpoint, we illustrate the application of the Programming-Based Optimal Control Mapping (P-OCM) method to an example of industrial relevance. A potential advantage of the P-OCM scheme stems from its capability of accurately incorporating the underlying continuous dynamics of a system (e.g., in our case, the battery state of charge) since the earlier design stage. Moreover, P-OCM has the potential of generating more scalable formulations (e.g., in terms of MILP) that can be efficiently handled by commercial solvers.

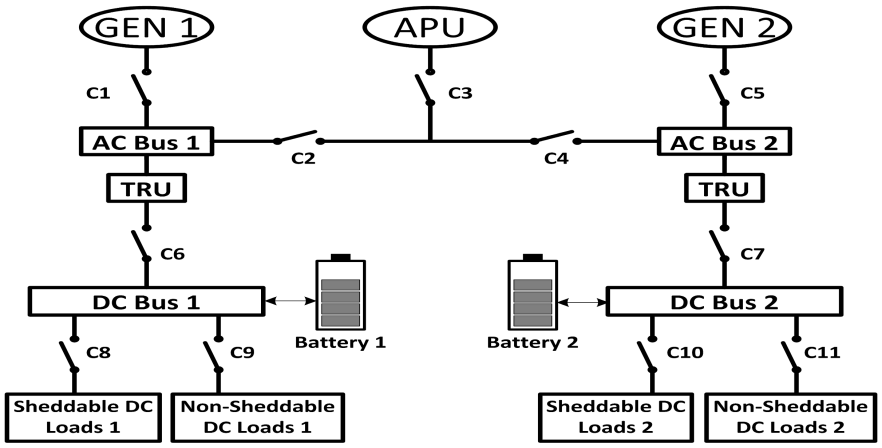


Figure 7.9: Single line diagram of the electric power system used in the formulation of the optimal load management problem. $C_i,\ \forall i \in \{1, \ldots, 11\}$ represent contactors.

After a brief description of the requirements for a load management system in an aircraft EPS in Section 7.3.1, Section 7.3.2 illustrates our hierarchical optimal load management approach, while Section 7.4 includes the formalization of the control problem on a sample power system topology, and simulation results showing its effectiveness.

7.3.1 Load Management Requirements

We have discussed the EPS system requirements, generally expressed in terms of safety, reliability and real-time performance properties, in Section 7.1.2. In this section, we refer to the simplified single line diagram in Figure 7.9, which we will use to describe the optimal load management problem formulation. Power system loads include sub-systems such as lighting, heating, motors, and actuators. A subset of loads are critical and cannot be shed, while others can be taken off-line in case of emergency. Current, voltage and contactor *sensors* are used to monitor the status of the system and to identify possible faults. The role of the EPS distribution system is to guarantee that loads are powered with the required power levels. Therefore, in addition to sensors, we recall that the EPS control system consists of *Generator Control Units* (GCUs) and *Bus Power Control Units* (BPCUs). Each GCU regulates the output voltage of a generator to meet the desired power level for a range of expected loads. Conversely, the BPCU ensures robust operation of the system for a number of failures in its components, by opening or closing the contactors to adequately reroute power to critical loads.

Based on this topology, we list two additional requirements that are relevant to the derivation of the optimal load management problem:

R1) *Load Shedding.* Sheddable loads are allowed to be shed if power supplies are insufficient, while non-sheddable loads must remain powered at all times. Typically, an electric

Table 7.3: Load Priority Table example.

Non-sheddable loads (W)		Sheddable loads (W)		Shed priority
Bus 1	Bus 2	Bus 1	Bus 2	
5000	4000	1000	1000	1 (shed first)
1000	1000	5000	2000	2
1000	1000	2000	2000	3
2000	2000	2000	5000	4
1000	11000	1000	1000	5
1000	1500	5000	4000	6
45000	2000	1000	1000	7
5000	39000	2000	3000	8
8000	10000	2000	2000	9
500	500	2000	2000	10 (shed last)

Table 7.4: Bus Priority Table example.

AC Bus 1	AC Bus 2	Priority
GEN 1	GEN 2	1 (most preferred)
GEN 2	GEN 1	2
APU	APU	3 (least preferred)

Load Management System (LMS) is responsible for protection and shedding of loads, by respecting a load priority list as the one shown in Table 7.3 for the SLD in Fig. 7.9. A higher shedding priority suggests a load that should be shed first.

R2) *Bus priorities.* Each bus may also have a priority list that specifies which generator should be used to provide power. If the first generator in the priority list is unavailable, then a bus should be powered from the second generator, and so on. An example of a bus priority table for the SLD in Fig. 7.9 is shown in Table 7.4.

Designing an efficient EPS controller is certainly a challenge, the main drawback of current implementations being the lack of optimality in load shedding and power source allocation. In the next section, we illustrate the new proposed architecture that addresses these issues.

7.3.2 Optimal Load Management System Architecture

We propose a hierarchical architecture that controls power source utilization, load shedding, contactor status and battery charge via two layers of controllers. Figure 7.10 shows a block diagram of the system (top), consisting of a *Low-Level LMS (LL-LMS)* and a *High-Level LMS (HL-LMS)*, and a timing diagram for its operation (bottom). The HL-LMS operates at a *slower clock* rate, with period T, and provides control optimality over a time horizon. The

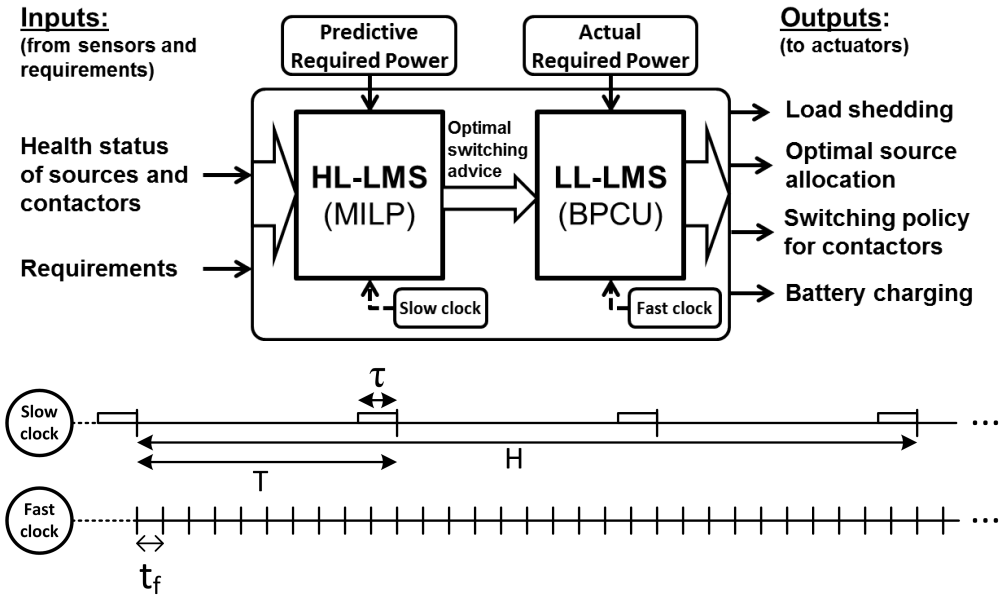


Figure 7.10: Block diagram of the proposed hierarchical load management architecture (top) and timing diagram for its operation (bottom).

LL-LMS operates at a *faster clock* rate with period $t_f < T$, and guarantees system safety by quickly reacting in the event of unexpected changes in loads or component failures.

The HL-LMS solves an optimal control problem at each step, using a receding horizon approach. The inputs to the HL-LMS are the required-power prediction for each bus over a time horizon of interest (H, in Figure 7.10), the health status (operational or faulty) of power sources and contactors, and the whole set of system requirements (i.e. the ones in Section 7.1.2, together with `R1` and `R2`). While each optimal control problem is solved for the larger time horizon H, only the initial samples of the solution (up to time T) are implemented and sent to the LL-LMS as *advice*.

The maximum computation time of the optimal control problem is assumed to be $\tau \leq T$. In fact, as discussed in Section 7.4.7, τ is usually much smaller than T in our application. However, to ensure more frequent updates to the HL-LMS, T can be chosen as $\max(\tau, t_f)$. Before the end of each slow clock cycle, by a time interval as long as τ, the optimal control problem is updated with the actual sensor readout on the status of sources, contactors and loads. A new solution is then computed and sent as advice.

The LL-LMS implements the BPCUs and, along with the GCUs, monitors the generator and contactor status more frequently (with a period t_f) to guarantee that each critical bus is powered at the desired voltage level (e.g. $T = 10t_f$ in Figure 7.10). At each time step, the LL-LMS actuates the advice from the HL-LMS only if this is feasible, given the *actual* status of contactors, power sources and loads. If this is not feasible, e.g. when an unforeseen fault in any component or an unpredicted change in power requirements are detected, the LL-

LMS reroutes power based on its predefined, worst-case control policy. Then, the LL-LMS keeps implementing its control policy until the next HL-LMS cycle, when the information on the failure is communicated to the HL-LMS, which updates the optimization problem with additional constraints that account for the failure. The new constraints will remain in place until the failure is resolved. The above scheme is always at least as effective as the LL-LMS controller in guaranteeing system safety, while implementing at the same time more efficient source allocation, load shedding, and battery utilization strategies in the absence of faults.

We assume that a BPCU is already available to implement the LL-LMS, by simply handling system faults without any concern on control optimality. Such a BPCU can be designed, for instance, using the RS-OCM approach in Section 7.2. The main focus of the next section is then the design of an optimal load management strategy for the HL-LMS and of its interface with the LL-LMS. Following our methodology, the predictive information about the *required power* for each bus, the *load characteristics*, the *generation profiles* and the component *connectivity* from the single-line drawing (SLD) provide an abstraction of the EPS *architecture*, including the controller, i.e. the architecture contract $\mathcal{C}_P \otimes \mathcal{C}_C$. On the other hand, the *safety* and *performance* requirements listed above originate a set of constraints capturing the system *function*, i.e. the application contract $\mathcal{C}_{HR}$. The resulting control policy leads to optimal load shedding and generator utilization, while satisfying system requirements as well as constraints on the battery state of charge captured by $\mathcal{C}_{HR}$. As described in Section 6.3, the optimization problem includes as constraints the conditions that must be enforced for the vertical contract $\mathcal{C}_{HR} \wedge (\mathcal{C}_P \otimes \mathcal{C}_C)$ to be consistent. In Section 7.4.7 we model the interaction between HL-LMS and LL-LMS for a set of simulation scenarios. A formal characterization of the interface between HL-LMS and LL-LMS in terms of vertical contract will be object of future work.

7.4 Optimal Load Management System Design

In this section, we describe the formulation of the optimal load management problem used to implement the HL-LMS, and provide simulation results to show its effectiveness. To detail the problem formulation, we use the SLD shown in Figure 7.9 as a running example. In Figure 7.9, AC buses are connected to transformer-rectifier units (TRU) to convert AC power to DC power. Each DC bus has sheddable and non-sheddable loads, as well as a battery set. The underlying assumption is that TRU cause no power loss, i.e. the generated power available at each AC bus is equal to the power delivered to the corresponding DC bus. Therefore, in the simplified network topology of Figure 7.9, each DC bus can be lumped together with the corresponding AC bus. A more complicate topology, such as the one in Figure 1.3, would entail minor modifications in our formulation. To determine the load and source allocation policy, we leverage power balance equations based on a simplified steady-state power flow analysis. Extensions of our approach to support optimal AC power flow analysis, including reactive power allocation, are out of the scope of this chapter, and will be considered as future work. The average power required by the loads in normal conditions,

Table 7.5: Nomenclature used for the optimal load management system formulation in this section (part 1).

Parameter	Definition
L^s_{ji}	Sheddable power sunk by load i of bus j
L^{ns}_{ji}	Non-sheddable power sunk by load i of bus j
n_j	Number of sheddable loads of bus j
N_j	Number of all loads of bus j
N^b	Number of buses
N^s	Number of power sources
$P^{\max}_j$	Maximum capacity of source j
$\overline{SoC}$	Maximum bound on battery state of charge
$\underline{SoC}$	Minimum bound on battery state of charge
T_s	Sampling time for HL-LMS optimization problem
T	Period of the slow clock
H	Prediction horizon of the optimal control problem
τ	Maximum computation time of the optimal control problem
t_f	Period of the fast clock
K	Parameter for battery charge dynamics
N^{as}	Number of allowed contactor switching events

as well as the load shedding priority, is given in Table 7.3, while the reference bus priority table considered in this section is in Table 7.4. A list of all variables used in this section is also available in Tables 7.5 and 7.6.

In the following, we detail the mathematical constraints expressing both the assumptions and guarantees of the vertical contract $\mathcal{C}_{HR} \wedge (\mathcal{C}_P \otimes \mathcal{C}_C)$. We categorize the constraints according to the main design concerns they reflect. As discussed in Section 7.1.2, such an encoding can also be facilitated by patterns related with these design concerns.

7.4.1 Load Modeling and Requirements

Non-sheddable and sheddable loads are denoted by L^{ns} and L^s, respectively. We model the required power at time t as the summation of contributions from different power sinks (loads), some of which are sheddable. Hence, for bus j

$$P_{\text{req}_j}(t) = \sum_{i=1}^{n_j} c_{ji}(t) L^s_{ji}(t) + \sum_{i=n_j+1}^{N_j} c_{ji}(t) L^{ns}_{ji}(t) \tag{7.1}$$

where, for $j = 1, 2, ..., N^b$, $P_{req_j}(t)$ is the total required power by all the electrical loads N_j connected to bus j. $L^s_{ji}(t)$ and $L^{ns}_{ji}(t)$ are the power sunk by load i connected to bus j at time t in the sheddable and non-sheddable case, respectively; n_j is the number of sheddable

Table 7.6: Nomenclature used for the optimal load management system formulation in this section (part 2).

Variable	Definition
γ_{ij}	Weight to penalize shedding of load j of bus i
λ_{ij}	Weight to capture source priority for source i to bus j
μ	Weight to penalize usage of an additional source
t_{chrg}	Time allowed for battery to reach $\underline{SoC}$
c_{ij}	Determines whether load i of bus j should be shed or not
P_{req_i}	Total required power of bus i
P_{sup_i}	Total power supplied to bus i
β_i	Power flow into/out of battery set i
$P_{i\text{to}j}$	Power delivered by source i to bus j
δ_{ij}	Determines connectivity of source i to bus j
α_i	Determines usage of source i
SoC_i	State of charge of battery set i
NoS_{ij}	Number of status transitions of contactor connecting source i to bus j

loads. Each coefficient $c_{ji}(t)$ for bus j and load i at time t is a binary decision variable specifying whether power $L_{ji}(t)$ must be supplied or it can temporarily be interrupted for sheddable loads, i.e.

$$c_{ji}(t) = \begin{cases} 1 & \forall\ i \in I_j^{ns}, \quad j = 1, 2, ..., N^b \\ \{0,1\} & \forall\ i \in I_j^{s}, \quad j = 1, 2, ..., N^b \end{cases} \tag{7.2}$$

where I_j^{ns} and I_j^s denote, respectively, the index set of non-sheddable and sheddable loads of bus j, and N^b is the total number of buses. Finally, we capture shedding priorities via the following constraints:

$$c_{j1}(t) \leq c_{j2}(t) \leq ... \leq c_{jN_j}(t) \quad \forall\ t \geq 0 \quad j = 1, 2, ..., N^b \tag{7.3}$$

so that loads get ranked based on their priority, e.g. for bus 1, since load 1 has the highest shedding priority, L_{11} must always be interrupted prior to L_{12} when the total supplied power is not sufficient. While (7.1) is a set of connectivity constraints in the guarantees of $\mathcal{C}_P \otimes \mathcal{C}_C$, constraints in (7.3) capture the load shedding priorities in $\mathcal{C}_{HR}$.

7.4.2 Source Allocation and Switching Policy

For each DC bus i, a power balance equation can be written as follows:

$$P_{\text{req}_i}(t) = P_{\text{sup}_i}(t) - \beta_i(t) \quad i = 1, ..., N^b \quad \forall\ t \geq 0 \tag{7.4}$$

where the amount of required power P_{req_i} from the loads, defined as in (7.1), is constrained to be equal to the amount of power supplied to bus i, $P_{\text{sup}_i}(t)$, decreased by the power used

for charging battery set i, denoted as $\beta_i(t)$. Therefore, when $\beta_i(t) > 0$, the battery set i is in a *charging* state, while $\beta_i(t) < 0$ implies that the battery set i is used to provide the power deficit. When no battery is present (as in AC buses), $\beta_i(t) = 0$ is enforced at all times.

The power supplied to bus i originates from one of the power sources, i.e. one of the engines or the APU. Assuming that there are N^b buses and N^s power sources, we enforce this constraint with the following equation:

$$\sum_{k=1}^{N^s} \delta_{ki}(t) P_{ktoi}(t) = P_{\mathrm{sup}_i}(t) \quad i = 1, ..., N^b \quad \forall\, t \geq 0 \tag{7.5}$$

where P_{ktoi} is the amount of power delivered by source k to bus i. Binary variables δ_{ki} determine which source should power which bus, so that $\delta_{ki}(t) = 1$ enforces that bus i is powered by source k at time t. Also, since no AC sources can be connected in parallel (per one of the requirements in Section 7.1.2), we need to enforce that each bus is powered by only one AC generator at every time. Hence,

$$\sum_{k=1}^{N^s} \delta_{ki}(t) = 1 \quad i = 1, ..., N^b \quad \forall\, t \geq 0 \tag{7.6}$$

Furthermore, we need to guarantee that the power available at each generator equals the power flow from the generator to the supported buses. This constraint can be enforced for power source j by the following equation

$$\sum_{k=1}^{N^b} \delta_{jk}(t) P_{jtok}(t) = \alpha_j(t) P_j^{\max}(t) \quad j = 1, ..., N^s \tag{7.7}$$

where $P_j^{\max}$ is the maximum capacity of power source j at time t (a known value), and $\alpha_j(t)$ is a binary variable denoting the usage of power source j at time t, i.e. $\alpha_j(t) = 1$ iff at time t source j is connected and used to power a bus. Clearly, since the number of active sources should be less than or equal to the number of buses, $\sum_{j=1}^{N^s} \alpha_j(t) \leq N^b$ must hold for all $t \geq 0$. However, due to the presence of constraints (7.6) and the selected cost function term (7.13), discussed below, this constraint becomes redundant and can be removed. Constraints in (7.4), (7.5), (7.7) are part of the $\mathcal{C}_P \otimes \mathcal{C}_C$ architecture contract guarantees, while constraints in (7.6) encode the non-paralleling requirement as a part of $\mathcal{C}_{RH}$.

7.4.3 Battery Dynamics and Requirements

For each battery set i, the normalized *State of Charge* (SoC) dynamics is captured using a simple model[2], as follows:

$$SoC_i(t+1) = SoC_i(t) + T_s \cdot K \cdot \beta_i(t). \tag{7.8}$$

[2]The ampere-hour (Ah) capacity of a battery depends on its temperature, rate of discharge, and age [191]. However, in this chapter we do not consider this level of detail for the battery model.

where T_s is the sampling time, $K > 0$ is a parameter constant for all battery sets, and $SoC \in [0, 1]$. $SoC = 1$ indicates a fully charged battery while $SoC = 0$ corresponds to a depleted battery. We would like to keep SoC within a safe interval, hence we enforce

$$\underline{SoC} \leq SoC_i(t) \leq \overline{SoC} \tag{7.9}$$

where $\underline{SoC}$ and $\overline{SoC}$, are the lower and upper limits on the SoC of batteries. If needed, constraint (7.9) can be enforced $\forall t \geq t_{\text{chrg}}$, where $t_{\text{chrg}} > 0$ is considered to allow time for battery charging up to the $\underline{SoC}$ bound (as done in the simulations of Section 7.4.7). Constraints in (7.8) form the battery behavioral model as a part of the guarantees of $\mathcal{C}_P \otimes \mathcal{C}_C$, while constraints as in (7.9) originate from the requirements on the battery state of charge, and are part of $\mathcal{C}_{RH}$.

7.4.4 Contactor Wear

It is also important to keep contactor switching activity as low as possible, to avoid contactor wear. For this purpose, the total number of status transitions (from open to close and *vice versa*) for a contactor that connects power source i to bus j over a time horizon of H time steps can be computed as:

$$NoS_{ij}(t^*) = \sum_{t=t^*}^{t^*+H\cdot T_s} |\delta_{ij}(t) - \delta_{ij}(t+1)| \tag{7.10}$$

where $|.|$ is the absolute value function. We then require

$$NoS_{ij} \leq N^{as} \quad \forall\ i = 1, ..., N^b, \qquad \forall\ j = 1, ..., N^s \tag{7.11}$$

where N^{as} is a safety threshold for NoS_{ij}. The above constraints will also be part of the guarantees of $\mathcal{C}_{RH}$.

7.4.5 Cost Function

In our formulation, we aim to minimize the total number (and duration) of load shedding (see requirement `R1`), as well as used generators. To achieve the first goal, we minimize the following function

$$\sum_{j=1}^{N^b} \sum_{t=t^*}^{t^*+H\cdot T_s} \Gamma_j^T [1 - C_j(t)] \tag{7.12}$$

where, $C_j(t) = [c_{j1}(t) \quad c_{j2}(t) \quad ... \quad c_{jn_j}(t)]^T$ is the vector of load coefficients for each bus j and $\Gamma_j = [\gamma_{j1} \quad \gamma_{j2} \quad ... \quad \gamma_{jn_j}]^T$ is a vector of weights used to penalize the act of load shedding for bus j. The components of Γ_j can be set to have the same value, or be used to capture the importance of each load. For instance, if sheddable load i is more important than j for AC bus k, we choose $\gamma_{ki} \gg \gamma_{kj}$. While the latter option provides more flexibility, it is not

essential to our formulation. In fact, the satisfaction of the priority tables for load shedding is already enforced by (7.3).

To achieve our second objective, i.e. minimize the number of generators utilized at all times, we augment the cost function with the following integral term

$$\mu \sum_{j=1}^{N^s} \sum_{t=t^*}^{t^*+H\cdot T_s} \alpha_j(t) \tag{7.13}$$

where μ is a constant weight parameter, which allows exploring the trade-offs involved in the multi-objective optimization problem.

Finally, we need to guarantee that the EPS obeys the bus priority table in Table 7.4 as far as possible (as per R2). To this aim, we enforce that the following integral expression be also minimized

$$\sum_{j=1}^{N^b} \sum_{t=t^*}^{t^*+H\cdot T_s} \Lambda_j^T \Delta_j(t) \tag{7.14}$$

where $\Delta_j(t) = [\delta_{1j}(t) \quad \delta_{2j}(t) \quad ... \quad \delta_{N^s j}(t)]^T$ is the source allocation variable vector for bus j and $\Lambda_j = [\lambda_{1j} \quad \lambda_{2j} \quad ... \quad \lambda_{N^s j}(t)]^T$ is a weighting vector that captures the source allocation priorities and penalizes the act of introducing new, unnecessary power sources in the first place. For instance, in the case of three power sources for bus 1, as in Figure 7.9, we can set $\lambda_{11} = 0$ (highest priority or no penalty), $\lambda_{21} \neq 0$ (second priority in the list) as a penalty for using the GEN 2 to power bus 1, and $\lambda_{31} > \lambda_{21}$ (last priority) as a penalty for using the APU. In general, we have $\lambda_{jj} = 0$ and $\lambda_{ij} \neq 0, \quad \forall\, i \neq j$. We capture the bus priority requirements using a penalty function instead of a hard constraint, since the HL-LMS policy is deemed as a recommendation in our formulation. When the total required power is within the ratings of more than one generator, the optimizer will not violate the priority table as it minimizes the overall cost. Conversely, when a power source is not able to meet the power requirement at a bus, a decision needs to be taken on whether a load should be shed or a new supply should be introduced in the network. Our formulation is flexible enough to allow exploration of the trade-offs involved in such a choice by modifying the weighting vectors.

7.4.6 Putting it All Together

Using (7.1)-(7.14), the optimal load management problem can be formulated as shown below, where $S = \{C_j(t), \Delta_j(t), \alpha_j(t), \beta_i(t), P_{\text{sup}_i}(t), P_{jtoi}(t)\}$ is the set of optimization variables, and constraints containing t should be evaluated at $t = \{t^*, t^* + T_s, ..., t^* + H \cdot T_s\}$. We observe that the predictive information about the *required power* for each load, $L^s_{ik}(t)$ and $L^{ns}_{ik}(t)$, and the *generation profiles* $P_j^{\max}(t)$ are known values, providing the assumptions for both $\mathcal{C}_P \otimes \mathcal{C}_C$ and $\mathcal{C}_{RH}$.

The result is a mixed integer nonlinear program because of constraints (7.15b) and (7.15c). However, we observe that every product $\delta_{ji}(t)P_{jtoi}(t)$ in (7.15b)-(7.15c) includes a binary variable $\delta_{ji}(t)$ and a real variable $P_{jtoi}(t)$ for which $0 \leq P_{jtoi}(t) \leq U_j(t)$ holds, where

Optimization problem for HL-LMS

$$\min_{S} \sum_{t=t^*}^{t^*+H\cdot T_s} \left\{ \sum_{j=1}^{N^b} [\Gamma_j^T(1-C_j(t)) + \Lambda_j^T \Delta_j(t)] + \mu \sum_{j=1}^{N^s} \alpha_j(t) \right\}$$

subject to:

$$\sum_{k=1}^{n_i} c_{ik}(t) L_{ik}^s(t) + \sum_{k=n_i+1}^{N_i} c_{ik}(t) L_{ik}^{ns}(t) = P_{\sup_i}(t) - \beta_i(t)$$
$$\forall i = 1, ..., N^b \tag{7.15a}$$

$$\sum_{k=1}^{N^s} \delta_{ki}(t) P_{\text{ktoi}}(t) = P_{\sup_i}(t) \quad \forall i = 1, ..., N^b \tag{7.15b}$$

$$\sum_{k=1}^{N^b} \delta_{jk}(t) P_{\text{jtok}}(t) = \alpha_j(t) P_j^{\max}(t) \quad \forall j = 1, ..., N^s \tag{7.15c}$$

$$\sum_{k=1}^{N^s} \delta_{ki}(t) = 1 \qquad \forall i = 1, ..., N^b \tag{7.15d}$$

$$SoC_i(t+1) = SoC_i(t) + \beta_i(t) \qquad \forall i = 1, ..., N^b \tag{7.15e}$$

$$\underline{SoC} \leq SoC_i(t) \leq \overline{SoC} \ \forall t \geq t_{\text{chrg}}, \forall i = 1, ..., N^b \tag{7.15f}$$

$$\sum_{t=t^*}^{t^*+H\cdot T_s} |\delta_{ij}(t) - \delta_{ij}(t+1)| \leq N^{as} \quad \forall i = 1, ..., N^b$$
$$\forall j = 1, ..., N^s \tag{7.15g}$$

$$\delta_{ij}(t) = \{0, 1\} \quad \forall j = 1, ..., N^b \quad \forall i = 1, ..., N^s \tag{7.15h}$$

$$c_{j1}(t) \leq c_{j2}(t) \leq ... \leq c_{jN_j}(t) \quad \forall j = 1, ..., N^b \tag{7.15i}$$

$$c_{ji}(t) = 1 \qquad \forall j = 1, ..., N^b \quad \forall\ i \in I_j^{ns} \tag{7.15j}$$

$$c_{ji}(t) = \{0, 1\} \qquad \forall j = 1, ..., N^b \quad \forall i \in I_j^s \tag{7.15k}$$

$$\alpha_i(t) = \{0, 1\} \qquad \forall i = 1, ..., N^s \tag{7.15l}$$

$U_j(t) = P_j^{\max}$, the maximum power capacity of source j, is known. Therefore, by exploiting these facts, we can reformulate the above problem as follows. We first introduce a set of new variables $\pi_{ji}(t) = \delta_{ji}(t) P_{\text{jtoi}}(t)$ to replace each product term in (7.15b)-(7.15c). Then, for each new variable $\pi_{ji}(t)$, we add the following constraints $\forall\ t \geq 0$:

$$0 \leq \pi_{ji}(t) \leq P_{\text{jtoi}}(t) \tag{7.16a}$$

$$P_{\text{jtoi}}(t) - U_j(t)(1 - \delta_{ji}(t)) \leq \pi_{ji}(t) \leq U_j(t) \delta_{ji}(t) \tag{7.16b}$$

Therefore, after these transformations, (7.15b) and (7.15c) can be replaced by the following constraints, enforced $\forall\ t \geq 0$

$$\sum_{k=1}^{N^s} \pi_{ki}(t) = P_{\mathrm{sup}_i}(t) \quad \forall i = 1, ..., N^b \tag{7.17a}$$

$$\sum_{k=1}^{N^b} \pi_{jk}(t) = \alpha_j(t) P_j^{\max}(t) \quad \forall j = 1, ..., N^s \tag{7.17b}$$

$$0 \leq \pi_{ji}(t) \leq P_{j\mathrm{to}i}(t) \tag{7.17c}$$

$$P_{j\mathrm{to}i}(t) - U_j(t)[1 - \delta_{ji}(t)] \leq \pi_{ji}(t) \leq U_j(t)\delta_{ji}(t) \tag{7.17d}$$

which turn the original problem into a MILP.

The above formulation can also support faulty scenarios with minor modifications. In fact, whenever a path between source j to bus i is not available, an extra constraint $\delta_{ji} = 0$ can be added to the problem, to account for either generator or contactor faults. This formulation provides the optimal solution while addressing the faulty event at the same time.

7.4.7 Experimental Results

To perform design space exploration, we implemented Holms [1], a framework for hierarchical optimal load management in aircraft power distribution systems. The MILP in Section 7.4 was formulated using Yalmip [129] and solved with Cplex [5] for a time horizon of 100 s and a sampling time T_s=1 s. The maximum number of allowed switching events over the time horizon was set to N^{as}=2. By using t_f =0.5 s for the LL-LMS period, $T = 10T_s$ =10 s for the update rate of the receding horizon optimization, and H =30 for the prediction horizon, we obtained 960 binary and 660 real decision variables. On a 4-core 2.67-GHz Intel processor with 3.86 GB of memory, the average and maximum solver times were 0.20 s and 0.29 s, respectively. Based on our experiments, we selected τ =1 s, which is consistent with the timing diagram in Figure 7.10. However, as we will show in Table 7.7, larger values for T and H can also be accommodated in our framework, as long as accurate predictions for the required power are available. Choosing H = 20 s leads to 640 binary and 440 real variables, reducing the average and maximum solver times down to 0.177 s and 0.282 s. Finally, $H \leq 10$ s was observed to drastically deteriorate the quality of the results, as HL-LMS did not have sufficient information on predicted power requirements to take an adequate decision ahead of time.

We assume that AC Bus 1 and 2 in Figure 7.9 have the power requirements shown in Figure 7.11. These profiles are selected to mimic a typical scenario for the whole aircraft mission, even if they are scaled to a smaller time span to speed up simulation. The maximum power capacity is assumed to be 100 kW for GEN 1 and GEN 2, and 104 kW for the APU. To demonstrate the advantage of a hierarchical, optimization-based approach, we present

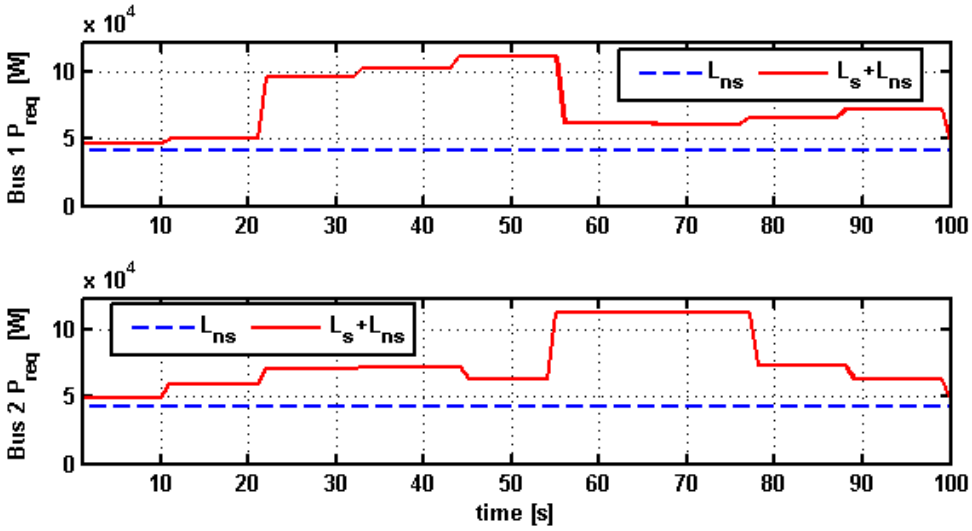


Figure 7.11: Power required by the AC Buses 1 and 2.

simulation results in the occurrence of a failure when only the LL-LMS is active and when both the LL-LMS and HL-LMS operate together.

7.4.7.1 Occurrence of Failure (LL-LMS Only)

Figure 7.12 shows the source allocation policy when only LL-LMS is active. A fault is introduced at $t = 45$ s for GEN 2, which keeps it in failure mode for the rest of the flight. Once LL-LMS detects the fault, it switches the power source for Bus 2 from GEN 2 to the APU. In fact, GEN 1, which has a higher priority in Table 7.4, is not enough to cover by itself the power demands of all the non-sheddable loads. To minimize load shedding, a peak of 112 kW in the required power could only be handled by leveraging the extra power supplied by the batteries, in addition to inserting the APU. However, LL-LMS makes no attempts at optimizing the number of used power sources at each time and can connect batteries only in the case of emergency. As a result, the number of shed loads, shown in Figure 7.13, is eventually higher than HL-LMS would propose.

7.4.7.2 Occurrence of a Failure (HL-LMS+LL-LMS)

To implement the HL-LMS, we set $\underline{SoC}$ and $\overline{SoC}$ to 0.25 and 0.75, and use the same values of γ for all the loads connected to each bus, since the satisfaction of their shedding priorities is already enforced via (7.15i) in the general problem formulation. Moreover, as a design choice, we set $\gamma_1 = \gamma_2 = 500$ and μ=10. As discussed in Section 7.4, we also select $\lambda_1 = [0 \quad 1 \quad 2]$ and, by symmetry, $\lambda_2 = [1 \quad 0 \quad 2]$.

Figure 7.14, Figure 7.15 and Figure 7.16 show source allocation, battery charge level and load shedding when GEN 2 fails at time $t = 45$ s. The HL-LMS is not able to predict such

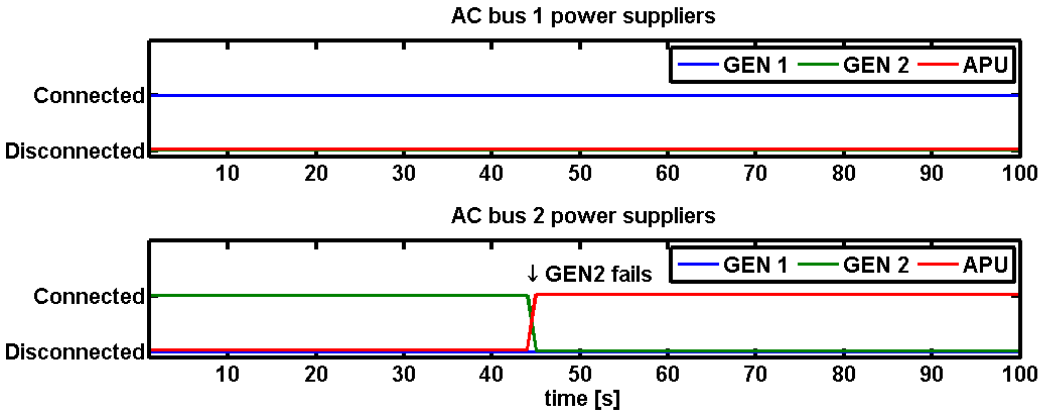


Figure 7.12: Power allocation in the case of failure under the operation of the low-level load management system (LL-LMS) only ($T_s = 1$ s, no battery utilization).

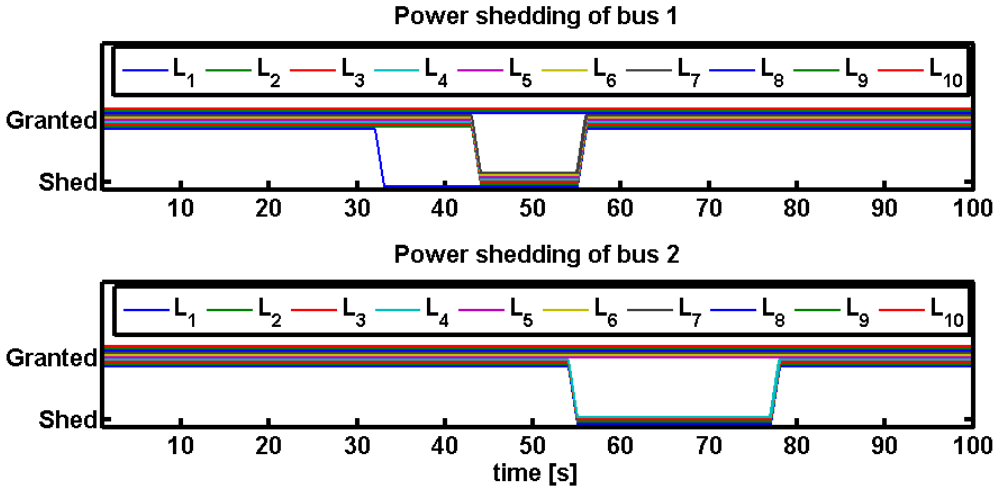


Figure 7.13: Load shedding in the case failure and under the operation of the low-level load management system (LL-LMS) only ($T_s = 1$ s, no battery utilization). Sheddable loads are labeled as $L_1, \ldots, L_{10}$.

a fault while computing the optimal control input for the interval $[40, 50]$ s. However, at time $t = 45$ s, the LL-LMS detects that GEN 2 has failed, discards the control input from the HL-LMS and uses its predefined control strategy to connect Bus 2 to the APU up to time $t = 50$ s. Only when the HL-LMS collects the actual health status of generators and contactors for the interval $[50, 60]$ s, it gets notified that GEN 2 has failed and is able to accommodate such a fault by incorporating the extra constraints ($\delta_{i2} = 0, \quad i = 1, ..., N^b$) to the MILP formulation. As a result, GEN 2 is no longer used during the mission.

Up to time $t = 44$ s, whenever DC Bus 1 requires more power than GEN 1 can provide,

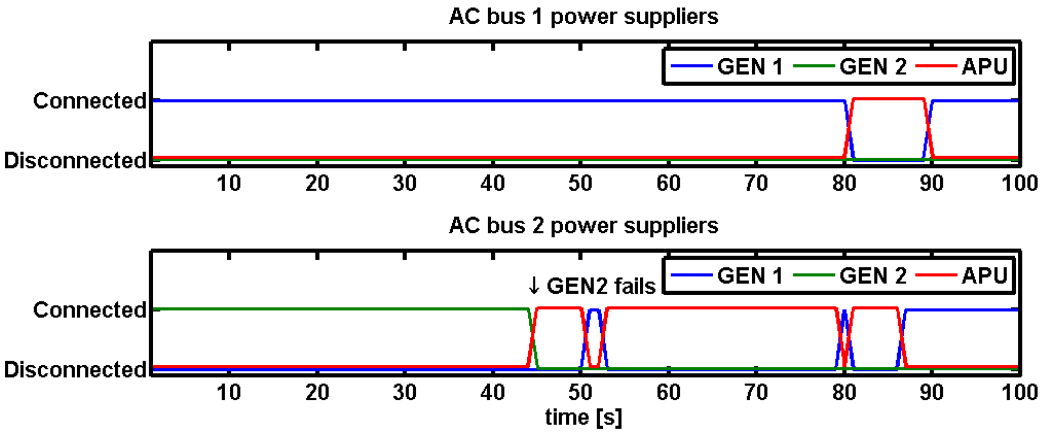


Figure 7.14: Power allocation in the case of failure under the operation of the hierarchical control scheme, including high-level and low-level management systems.

batteries are used as a backup supply. This is no longer the case after GEN 2 fails. Since no battery charge control is implemented at the LL-LMS level, the only possible solution is to shed the loads from Bus 1 to decrease its power requirements. Such loads are then powered back as soon as the advice from the HL-LMS is implemented again.

7.4.7.3 Discussion

Simulation results confirm the effectiveness of our algorithm, capable of providing an optimal policy that satisfies both EPS safety and performance requirements. To explore the impact of γ and μ parameters on the result, we performed a set of tests for different values of these parameters. As expected, the smaller the values of γ_1 and γ_2, the higher is the number of shed loads and the smaller is the number of used generators over time. On the other hand, smaller values for μ tend to encourage the use of more generators than the ones strictly needed because of power requirements.

To compare the performance of the hierarchical control architecture with the one of a conventional controller (LL-LMS only) in terms of load shedding, we define a normalized shedding index, which quantifies the percentage of shed loads over the duration of a mission, and is based on the cost term in (7.12):

$$I_{\text{shed}} = \frac{\sum_{j=1}^{2} \sum_{t=0}^{100} \Gamma_j^T [1 - C_j(t)]}{\sum_{j=1}^{2} \sum_{t=0}^{100} \Gamma_j^T \mathbf{1}} \cdot 100\%, \tag{7.18}$$

where $\mathbf{1} \in \mathbb{R}^{N_j}$ is a vector of ones. In our simulations, I_{shed} decreases from 9.2% for LL-LMS to 1.7% for the hierarchical controller, which means a 5-fold improvement in the latter case.

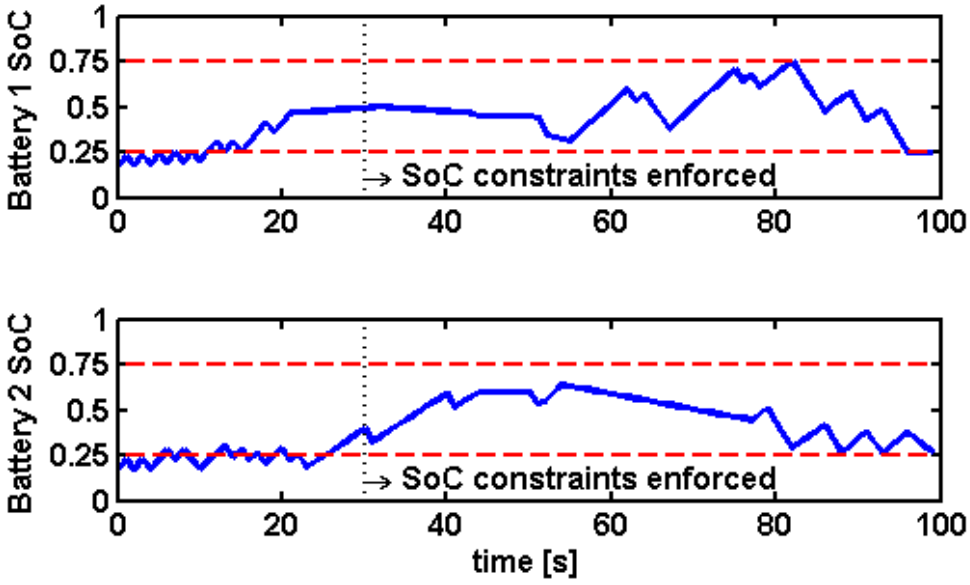


Figure 7.15: Battery charge level in the case of failure under the operation of the hierarchical control scheme ($T_s = 1$ s, $\underline{SoC} = 0.25$, $\overline{SoC} = 0.75$, $t_{\text{chrg}} = 30$ s).

Table 7.7: Number of optimization variables and solver time for a 2-bus 3-generator electric power system, when the time horizon increases.

Prediction horizon (H)	10	20	50	100
Number of opt. var.	430	860	2150	4300
Solver time (s)	0.3	0.19	1.25	25

Similarly, a source utilization index can be defined based on (7.14):

$$I_{\text{source}} = \sum_{j=1}^{2} \sum_{t=0}^{100} \Lambda_j^T \Delta_j(t), \tag{7.19}$$

which quantifies the cost associated to the usage of power sources over the duration of a mission. In our simulation, I_{source} decreases from 110 (LL-LMS only) to 76 (LL-LMS + HL-LMS), which is a 31%-reduction in the hierarchical control case.

To test the *scalability* of the proposed framework, we performed optimizations with different time horizons. No substantial improvement in the quality of the solution was observed for $H > 30$ in our experiments, in spite of the larger computation time. However, as evident from Table 7.7, even problems with thousands of variables can be solved in a few seconds using the proposed formulation. Finally, we also implemented controllers for EPS topologies with a larger number of generators and loads, in which the number of buses and contactors

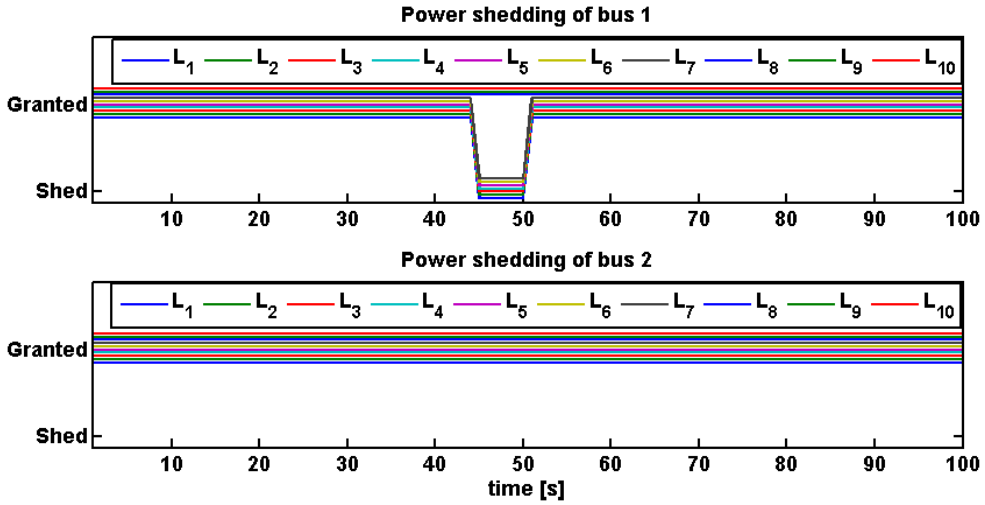


Figure 7.16: Load shedding in the case of failure and under the operation of the hierarchical control scheme. Sheddable loads are labeled as $L_1, \ldots, L_{10}$.

Table 7.8: Number of optimization variables and solver time for H=30 (B and G stand for the number of Buses and Generators, respectively).

Number of nodes	Number of opt. var.	Solver time (s)
B=4, G=3	binary: 960, real: 660	ave: 0.20, max: 0.29
B=10, G=5	binary: 2300, real:1650	ave: 2.10, max: 2.22
B=20, G=10	binary: 6100, real: 6300	ave: 6.87, max: 7.0

is also increased proportionally. The results in Table 7.8 show that, for a realistic number of generators (normally less than 10), computation times stay largely compatible with the timing assumptions needed for the correct operation of the proposed hierarchical scheme.

7.5 Aircraft Air Management System Design Overview

Figure 7.17 shows the simplified architecture of a Pressurization and Air Conditioning Kit (PACK) of an aircraft air management system (AMS). Engine bleed air enters the PACK through *Valve 1*, and gets partitioned into two flows, one passing through the bypass *Valve 2*, and the other passing through the heat exchanger (*HX*). Then, the flows recombine in a *Mixer* and enter the aircraft cabin. In a PACK, the cabin pressure is typically controlled by a set of electrical compressors (not shown in Figure 7.17), while the cabin temperature is regulated via the heat exchanger and, possibly, expansion cooling across one (or more) turbines [147]. In our simplified diagram, *Valve 1* is responsible for the flow-rate W_i into

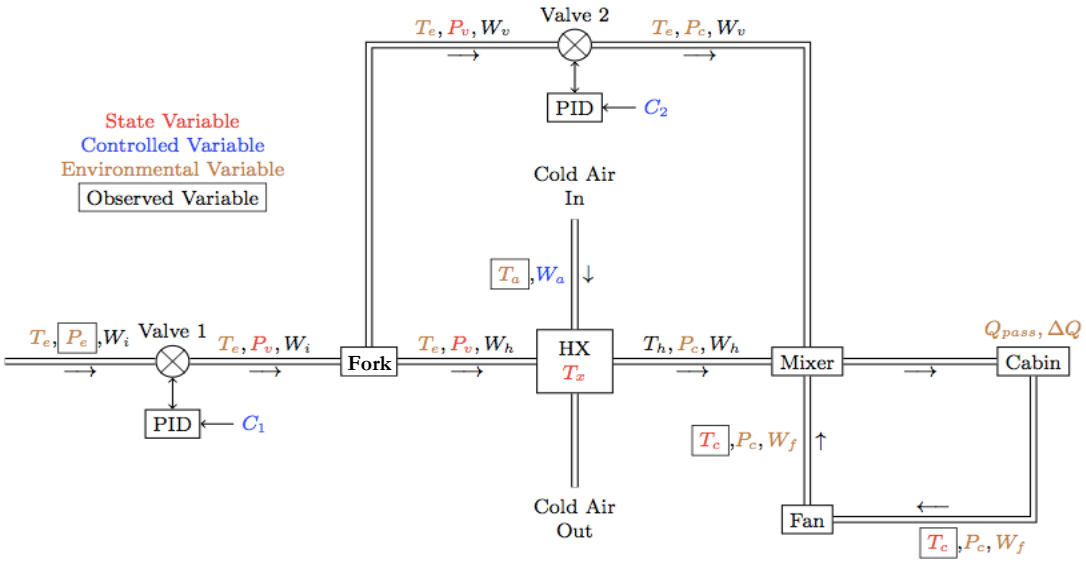


Figure 7.17: Simplified architecture of a Pressurization and Air Conditioning Kit (PACK) of an aircraft air management system.

the system, whereas *Valve 2* directly influences the cabin temperature by controlling the fraction of inflow that is cooled in the heat exchanger by the cold ambient mass flow W_a. The AMS needs to be designed to supply sufficient pressure, and fresh oxygen, to the cabin at a comfortable temperature and humidity, while being resilient to faults, such as freezing or warping of critical components.

In this section, we show how the methodology of this dissertation, and its main steps, can also be applied to this case study, by adequately adapting them to the problem at hand. Differently than the EPS design in the previous sections, in this example, we assume that the plant topology, hence its reliability, is fixed. On the other hand, the controller has a hierarchical structure, where a high-level supervisor decides the AMS operating mode (e.g. climbing, cruising) and provides the appropriate set points to the lower-level controllers, implemented using model predictive control or proportional-integral-derivative (PID) architectures. Therefore, our goal is to determine the plant sizing parameters and the control strategy for valves 1 and 2 and flow-rate W_a to satisfy a set of top-level requirements, under the assumptions that the temperature and pressure of the bleed air are piecewise constant (possibly changing over the duration of a flight) while the temperature of the cold air flowing into the heat exchanger is affected by the altitude of the airplane and the time of the day.

Since the PID controllers are right at the interface with the plant, we need to incorporate their effect as we determine the plant sizing, in order to enable independent implementation of plant architecture and control algorithm. Moreover, to accurately capture the plant dynamics, we resort to a continuous, non-linear optimization scheme, where the sys-

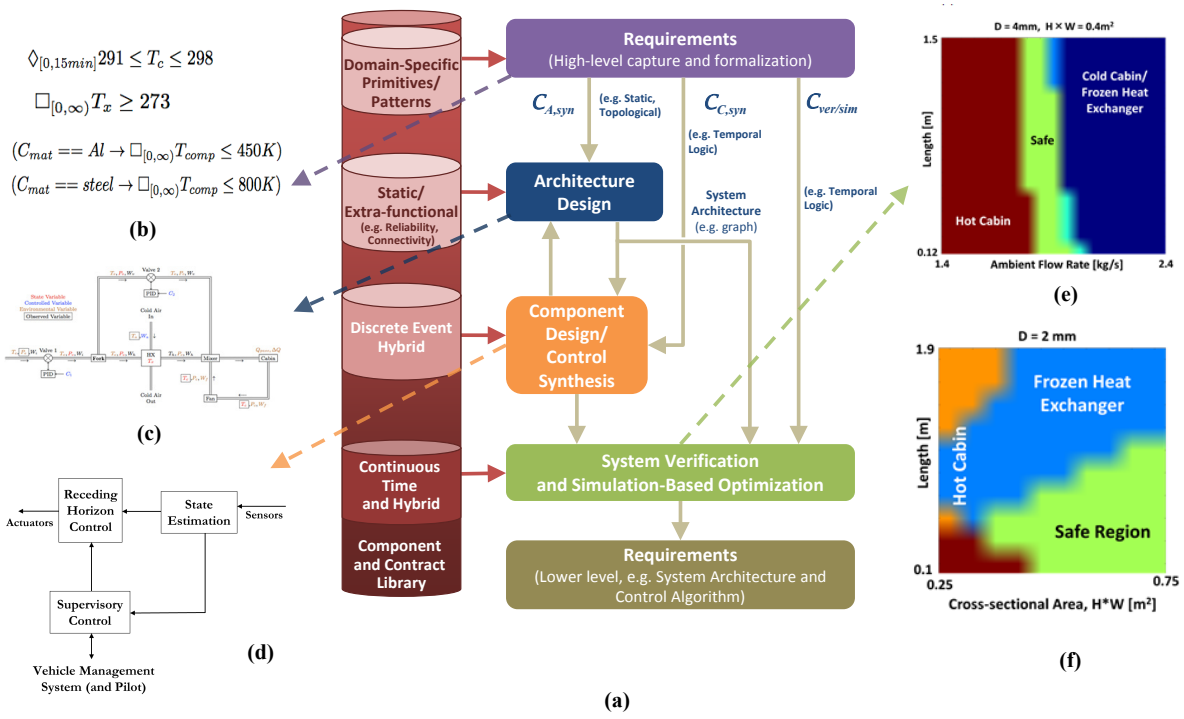


Figure 7.18: Representation of the proposed design flow as applied to an aircraft air management system.

tem steady-state and transient performance is evaluated from simulation traces using an acausal, parametrized model of the plant in closed-loop with its controllers, implemented in MODELICA. The *system requirements* include, among others, bounds on the desired cabin temperature (e.g. within 291 K and 298 K), the maximum time allowed before reaching the steady state, the minimum heat exchanger outlet temperature to avoid freezing.

The overall design flow is represented in Figure 7.18. In the *architecture design* step, top-level requirements are then translated into STL contracts and their satisfaction is checked on the simulation traces, while searching the space for an optimal plant sizing that minimizes the overall volume and component cost. Architecture design can then been seen as instance of the "lazy" optimized mapping approach, where the role of the oracle is performed by the simulation engine. In the *control design* step, for a given plant architecture, the same requirements can be used, together with a plant model based on ordinary differential equations, to synthesize both the DE supervisory control and the lower-level MPC schemes, using, respectively, reactive synthesis from temporal logic contracts and programming-based optimized control mapping, both discussed in Chapter 6. The result will be a fully sized MODELICA model of the plant, and a hybrid model of the overall controller.

As an example, Figure 7.19 (a) shows design exploration results capturing the impact

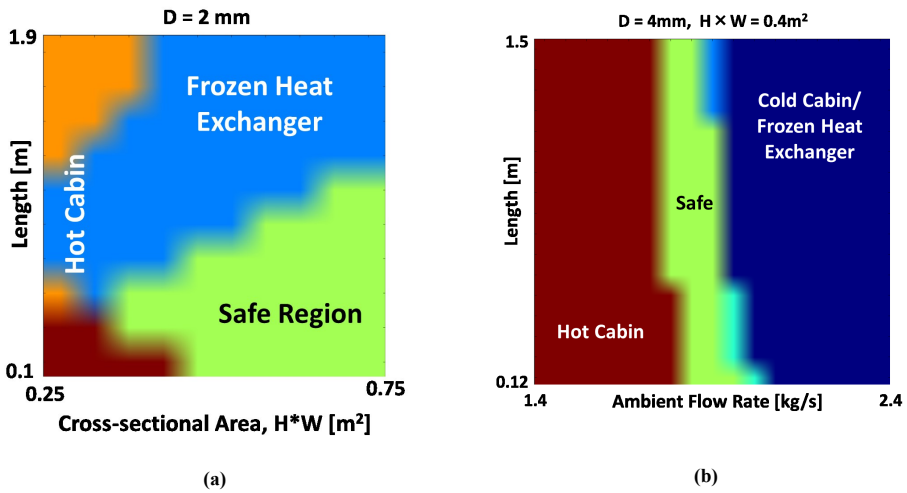


Figure 7.19: Air management system design space exploration example.

of the heat exchanger geometry, i.e. cross-sectional area and length, on the system correctness at steady state. Figure 7.19 (b) shows instead the feasible design space in a co-design scenario, when the steady state can be selected together with the plant sizing, thus making it possible to explore trade-offs between plant sizing and control set points. In both the figures, the safe regions, where no requirements are violated and the system behaves correctly, are highlighted in green. Each exploration run, including 250 simulations, required approximately two minutes on an Intel Core i5 2.53-GHz processor with 4-GB memory.

7.6 Conclusions

We provided a detailed example of application of the methodology proposed in this thesis to the design of aircraft power distribution systems, including the following steps: top-level requirement formalization using a pattern-based contract specification language, independent refinement of system architecture and control strategy via vertical contracts, optimized selection of reliable and cost-effective architectures, and optimized control mapping via reactive synthesis from linear temporal logic contracts combined with simulation-based optimization subject to the satisfaction of signal temporal logic contracts. We then addressed the problem of optimal load management for aircraft electric power distribution, offering a hierarchical scheme, Holms, to coordinate load shedding, source allocation and battery utilization, by solving an efficient mixed integer-linear program within a receding horizon approach. In addition to guaranteeing system safety, our hierarchical architecture shows substantial performance improvements with respect to the state of the art in terms of percentage of shed loads and number of utilized sources. Moreover, simulation results show that the optimal control problem scales reasonably in the context of the selected application.

While certain formalisms following a "declarative" style, such as temporal logic and arithmetic constraints on Boolean and real numbers, have shown to be promising for requirement formalization and the deployment of correct-by-construction synthesis and mapping techniques, the direct adoption of reactive synthesis for industrial-scale problems seems difficult, at least with the current tools, because of its computational complexity. On the other hand, the Programming-Based Optimized Control Mapping scheme shows promise of more accurately incorporating the underlying continuous dynamics of a cyber-physical system since the earlier design stages, albeit for a finite time horizon, and generating more scalable control problem formulations that can be efficiently handled by commercial solvers.

Altogether, in this chapter, we demonstrated a combination of the strategies proposed in this dissertation to overcome design complexity, including:

- Leveraging efficient, compositional approximations of complex system metrics (e.g. reliability) to perform system-level design exploration in the concept design phase, with rigorous, quantitative bounds on the approximation errors;
- Developing efficient mapping algorithms that can traverse a large design space by decomposing complex decision problems into a combination of smaller tasks, each carried out by the most appropriate tool (e.g. the Integer-Linear Programming Modulo Reliability algorithm);
- Addressing different design concerns (e.g. safety versus real-time performance) by developing different abstractions (e.g. discrete-event versus hybrid) formally related by vertical contracts, as in the Reactive Synthesis-Based Optimized Control Mapping method;
- Leveraging hierarchically organized, pre-characterized library of contracts to speed up verification and mapping tasks using on-the-fly abstractions and refinements based on library mappings, as in the library-based scalable refinement checking algorithm;
- Developing efficient optimization-based formulations of control design problems that can be solved with a receding horizon approach and scale better than reactive synthesis methods, as in the Programming-Based Optimized Control Mapping method used in HOLMS.

Finally, we concluded the chapter by showing how systems exposing apparently different behaviors, such as an aircraft air management system, can still be designed, with appropriate adaptations, using a combination of the methods above.

Chapter 8

Conclusions and Future Work

In this chapter, we summarize the main contributions of this dissertation and suggest promising research directions for future work.

8.1 Conclusions

By "instrumenting" our planet with information technology based devices, cyber-physical systems (CPS) have the potential to provide a variety of novel services with a societal-scale impact, radically influencing how we deal with crucial problems, such as national security and safety, energy management, environment control, efficient and reliable transportation, and affordable health care. The complexity and heterogeneity of CPS is, however, increasing the design and verification challenges. In several industries, the development of these systems becomes increasingly more expensive and time-consuming. Serious vulnerabilities due to unwanted interactions found late in the development process tend to affect the realization of these systems, and are possibly insurmountable with the methodologies in use today. While compositional approaches, stepwise refinements of requirements, and reuse of pre-designed components are deemed as instrumental to reduce design cost and make system integration affordable, the definition of a principled, compositional methodology that can support the creation and exchange of model libraries, tool interoperability (allowing models to be analyzed or optimized by multiple tools), and multi-view modeling (rigorously combining models that represent different aspects of a design, e.g. a control-logic model with an energy-consumption model) is an open problem.

The research in this dissertation seeks to advance the state of the art in compositional design methodologies by spanning the whole spectrum from theoretical foundations, in numerical and formal methods, to practical tools for system analysis and design. We introduced a methodology that addresses the complexity and heterogeneity of cyber-physical systems by leveraging the concepts of horizontal and vertical contracts to formalize the design process in a hierarchical and compositional way, and interconnect different modeling, analysis and synthesis tools, to ensure quality and correctness of the final result. We presented theoretical

results, methods, and algorithms showing that it is indeed possible to provide the formal foundations for such a methodology, covering both horizontal and vertical compositions. To illustrate the application of the methodology, we used concrete examples from controller design in aircraft electric power and air management systems.

We adopted assume-guarantee (A/G) contracts as a rich and flexible contract model to provide the formal foundations of our methodology, and carved out of the general theory a concrete contract framework that can fully support multi-view and multi-layer design flows. A first requirement of such a contract framework is to support compatibility and consistency checking for systems with uncontrolled inputs and controlled outputs, a widely used abstraction in the design of networked controlled systems. To address this requirement, in Chapter 3, we drew inspiration from interface theories, another class of specification theories, which shares the same overall objectives as contracts, but is based on different mathematical underpinnings. Specifically, we established a link between the theory of A/G contracts and the one of relational interfaces, shedding light on some of their key features for system specification, early detection of incompatibilities, and use of abstraction-refinement. We proposed a natural transformation from interfaces to linear temporal logic contracts, and analyzed differences and correspondences between key operators and relations in the two theories (i.e. composition, refinement and conjunction), by studying their preservation properties under the proposed transformation. We showed that the transformation preserves refinement, but does not generally preserve serial composition, interface compatibility, and conjunction. Then, we proposed a new projection operator on contracts that captures the distinct nature of inputs and outputs during hiding, thus enabling preservation of the semantics of interface composition and compatibility.

A second requirement for an effective contract framework for cyber-physical system design is to encompass richer refinement relations between different representations of the design, including synthesis and optimized mapping between abstraction levels described by heterogeneous architectures and behavior formalisms. To address this requirement, we formalized the concepts of heterogeneous refinement and vertical contracts to deal with hierarchies of models characterized by both semantic and structural heterogeneity, i.e. using different formalisms and architectural decompositions.

In Chapter 4, we detailed the structure of our methodology as a layered process including three main steps, in which high-level specification contracts are mapped into lower-level aggregations of contracts out of a pre-characterized library. At each step in our methodology, top-level requirements are captured as contracts using a controlled specification language based on pre-defined patterns. Patterns facilitate the translation of contracts into mathematical formalisms. First, a set of steady-state, structural and reliability requirements are mapped into a high-level system architecture (architecture design); then, a set of functional, safety and reliability requirements are mapped into a lower-level, discrete, dynamical representation of the system (control design). Finally, real-time performance requirements are mapped into an aggregation of higher-fidelity hybrid components by a simulation-based optimization step, which is also used to verify all the assumptions made at the earlier design stages. We presented a proof-of-concept framework, CHASE, which uses A/G contracts to

coordinate different mapping tools and enable rigorous analysis of complex behaviors in a way that is practically usable by system engineers.

We the focused on the description of different mapping algorithms, and their demonstration on industrial case studies. Chapter 5 showed that scalable system-level design exploration of CPS architectures subject to safety and reliability constraints is possible by combining optimization methods with compositional approximations of complex system metrics (e.g. reliability), for which rigorous, quantitative error bounds can be proved. Moreover, we demonstrated how concepts from formal methods can be effectively applied to traverse large design spaces by decomposing complex decision problems into smaller tasks, each carried out by the most appropriate analysis technique and tool. In particular, we introduced, characterized, and demonstrated two efficient optimization-based algorithms for the optimal selection of cyber-physical system architectures. We implemented both the algorithms in the ArchEx framework, and demonstrated their effectiveness on the design of aircraft power system architectures. Since generating exact reliability constraints by failure enumeration is expensive, the Integer-Linear Programming with Approximate Reliability (ILP-AR) algorithm generates larger, monolithic problem instances using efficient but approximate constraints computations. Conversely, the Integer-Linear Programming Modulo Reliability (ILP-MR) algorithm breaks the complex optimal architecture selection task into a sequence of smaller optimization tasks interleaved with exact reliability checks. By relying on efficient mechanisms to prune out large portions of the discrete space that are inconsistent with the reliability requirements, ILP-MR outperformed ILP-AR on large problem instances.

In Chapter 6, we proposed two methodologies for systematic design of control protocols using contracts: Reactive Synthesis-Based Optimized Control Mapping (RS-OCM) and Programming-Based Optimized Control Mapping (P-OCM). In the absence of tools that can fully support the design and verification of distributed control systems, RS-OCM addresses different design concerns (e.g. safety versus real-time performance) by developing appropriate abstractions (e.g. discrete-event versus hybrid) formally related by vertical contracts. A subset of the design requirements can be encoded into a linear temporal logic contract for which reactive synthesis is tractable and can generate a discrete-event representation of the controller. This abstract controller model satisfies its contract by construction, and is then refined into a higher-fidelity model using simulation-based optimization methods subject to the satisfaction of signal temporal logic contracts. On the other hand, P-OCM relies on an optimization-based formulation of the control design problem over a bounded time horizon, and its online solution within a receding horizon approach, which tends to scale better than reactive synthesis methods.

In the context of contract-based verification of distributed controllers, we addressed the problem of performing contract refinement checking efficiently, and presented an algorithm that leverages the contract library, enriched with refinement assertions, to break the main verification task into a set of smaller tasks. The application of the proposed algorithm to verify controllers for aircraft electrical power systems showed up to two orders of magnitude improvement with respect to standard implementations, e.g. based on solving LTL satisfiability problems with state-of-the-art tools. Since the library characterization process must be

performed only once, outside of the main design flow, the benefits of having a richer library in terms of refinement assertions largely repay the overhead of building it. 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.

In Chapter 7, the effectiveness of a contract-based design flow was demonstrated, for the first time, on an industrial case study, i.e. system-level design of an aircraft electrical power system, including several techniques discussed in this thesis: top-level requirement formalization using patterns, compositional refinement of system architecture and control algorithm, optimized selection of reliable and cost-effective power system topologies, and reactive synthesis based optimized control mapping. In our case study, formalisms following a "declarative" style, such as temporal logic and arithmetic constraints on Boolean and real numbers, proved to be promising for requirement formalization and deployment of correct-by-construction synthesis and mapping techniques. While the direct adoption of reactive synthesis for industrial-scale problems tends to be impractical today, the RS-OCM and P-OCM paradigms help alleviate the computational complexity issues, by relying, respectively, on the separation of design concerns between different abstraction levels, and on efficient optimization-based formulations.

We demonstrated the capabilities of optimization-based control design methods on the problem of optimal load management in aircraft power systems. We developed a hierarchical control architecture in which a high-level load management system (HL-LMS) coordinates load shedding, source allocation and battery utilization, by solving an efficient mixed integer-linear program within a receding horizon approach. The result of the optimal control problem is offered as advice to a low-level load management system (LL-LMS) that can directly actuate the EPS contactors, and decides to implement the advice from the HL-LMS only when it is safe. In addition to guaranteeing system safety, our hierarchical architecture showed substantial performance improvements with respect to a conventional one, based on just an LL-LMS, in terms of percentage of shed loads and number of utilized sources.

An overview of the proposed design flow, as applied to an aircraft air management systems, marked the conclusion of Chapter 7, showing how the design of systems characterized by behaviors and requirements apparently different in nature can still be addressed, with a few adaptations, within the same framework.

8.2 Future Work

Inspired by the design examples in this thesis, we envision a scenario in which a design management feature that we call a front-end *orchestrator* directly interacts with the designer, helps coordinate the set of back-end specialized tools, and consistently processes their results. For such an orchestrator to be developed, it is essential to develop tools to effectively guide designers towards requirement formalization, and *algorithms* that can maximally leverage the modularity offered by contracts, by directly working on their representations to perform

compatibility, consistency and refinement checks on system portions of manageable size and complexity. Moreover, these algorithms should take advantage of any violation of the design constraints, i.e. a "counterexample" for system correctness, to provide meaningful *feedback* to the designer, and possibly set up *learning* strategies to refine or augment both the contract assumptions and guarantees until a final implementation is reached. In the following, we broadly categorize the future research directions entailed by the above scenario into three major areas: theory, algorithms, and applications.

8.2.1 Theory

Our work has pioneered a new research direction aiming to investigate the relation between different contract and interface theories, and their mathematical foundations, with the ultimate goal of developing a truly unifying compositional framework for system design. We expect more efforts will appear in this direction. For example, future extensions of the results presented in Chapter 3 include studying the properties of the proposed transformation with respect to feedback composition, which is not always defined for relational interfaces, as well as its generalization to the theory of interface automata. We are also interested in investigating a reverse transformation that maps A/G contracts into relational interfaces, which requires extending the latter with liveness properties. A recent effort has been reported in this direction, extending the expressiveness of relational interfaces to also allow formulas in higher-order logic and temporal logic [166].

8.2.1.1 Analog and Hybrid Contracts

As demonstrated in this thesis, a "natural" use of contracts in CPS design is to govern the horizontal composition of the cyber and the physical components and to establish the conditions for correctness of their composition. While we offered a number of strategies to model and analyze such a composition, and support heterogeneous refinements including hybrid behaviors, we observe that a comprehensive interface theory for continuous-time, infinite-state-space dynamic systems is currently lacking.

A major set of challenges in this domain stems from the difficulty of devising the most suitable abstractions for the verification of continuous-time continuous-valued systems, which can be consistently related to simulation models as well as models for semantic analysis (e.g. structural and performance models), possibly including reduced order models, or coarser, discretized and quantized approximations of physical behaviors for efficient system-level design space exploration. We have recently reported a first step in this direction in the context of mapping algorithms for heterogeneous architectural spaces [80], by leveraging extensions of the framework proposed in this thesis and our previous formulation for analog and mixed-signal (AMS) integrated circuit design [152].

8.2.1.2 Stochastic Contracts

Several parameters impacting the behavior of CPS are subject to variability due to manufacturing tolerances, usage and faults. Moreover, the models that are normally used to design multi-physics systems inevitably introduce inaccuracies [155]. As a consequence, robust system design often implies costly Monte Carlo simulations or over-design to guarantee large safety margins. Contract frameworks for stochastic system design are still in their infancy. However, the importance of providing a better support for reasoning about the probabilistic properties of systems and the deployment of robust design techniques cannot be overemphasized. In this context, advancing the state of the art in compositional approaches for *stochastic systems* and *stochastic contract frameworks*, e.g. by building on the works by Kwiatkowska et al. [119], Caillaud et al. [49], and Gössler et al. [87], is deemed as essential to improve on the scalability of stochastic analysis and synthesis techniques [117, 118], and make their adoption actually feasible in current design flows.

8.2.2 Algorithms

8.2.2.1 Requirement Analysis and Validation

In the context of requirement validation, a future research direction is to investigate efficient algorithms to implement the key operations and relations of the contract algebra for specification formalisms that are richer than LTL, such as (fragments of) STL and HRELTL. A framework for refinement checking of HRELTL contracts has been recently reported [61, 58]. On the other hand, efficient algorithms for contract compatibility or consistency checking, including the implementation of the assumption-projection operator in Chapter 3, are missing. In this respect, a major challenge is the computational complexity of quantified logics.

Cyber-physical system verification is expected to largely benefit from the development of efficient and accurate decision procedures and Satisfiability Modulo Theory (SMT) solvers that allow reasoning about Boolean combinations of linear and non-linear constraints over the reals. To implement efficient solvers over the reals, a promising approach is to decompose complex decision problems into smaller tasks, and leverage a "lazy" combination of formal methods and optimization techniques to tackle them, as we proposed for CalCS [150], an SMT solver using duality theory to reason about nonlinear convex constraints, and further demonstrated in this thesis in the context of different mapping problems. More recently, we have successfully applied the same paradigm to the problem of state reconstruction from corrupted sensor measurements in CPS under attack [186, 187, 185].

Finally, the recent advances in machine learning and data mining technologies can also be used to improve requirement validation in different ways. Results from system identification and statistical learning theory can be used to generate (and validate) models of components and subsystems for fast simulation and architecture exploration from design and measurement data to enrich and validate the contract library. However, in addition to modeling and simulation, data can be used to also improve other design tasks, such as synthesis or formal verification. In fact, both formal verification and synthesis algorithms generate *per*

se a substantial amount of data, such as "certificates" and "counterexamples," e.g. in terms of traces or scenarios. Then, novel diagnostic tools can be developed by leveraging this information to provide useful feedback to the designer, e.g. by suggesting modifications to the original specification when it is unrealizable, as recently proposed in the context of requirement mining [127, 126, 105, 128], and further generalized by the "sciduction" paradigm [181], advocating the integration of traditional, deductive methods with inductive inference (learning from examples), using hypotheses about the system structure, to tackle major challenges in formal verification. Specifically, requirement mining is an emerging field with several usage scenarios that can be an important complement to the methodology discussed in this thesis. In addition to helping create abstracted views of systems and components, mined requirements may be used to gain better understanding of legacy models or code, and may also help enhance the process of bug-finding through simulations [105].

8.2.2.2 Synthesis and Optimized Mapping for Architecture and Control

The benefits of horizontal and vertical contracts and the optimized mapping techniques demonstrated in this thesis can also be further explored to improve on the scalability of our architecture synthesis and reactive synthesis approaches. Possible research directions include:

- *Optimized selection and sizing of CPS architectures.* The ILP-AR and ILP-MR algorithms proposed in this thesis for efficient exploration of system architectures can be further generalized to encompass a larger set of requirements, and finally stand out as two complementary paradigms to traverse complex, heterogeneous design spaces for the implementation of the next generation of system-level design frameworks. As a future work, we plan to further investigate the generalization of these approaches to support a broader category of architectures (e.g. power grids, communication networks) and design concerns, such as the impact of system dynamics and transients, which may require the integration of discrete decision procedures and optimization frameworks with algorithms and tools for formal and semi-formal verification of hybrid systems. We have recently taken a step in this direction by proposing a "lazy" combination of a discrete and a continuous optimization engine based on continuous-time simulation for optimal sizing of CPS architectures [80].

- *Optimization-based synthesis.* More systematic ways can be devised to include cost (or reward) functions in temporal logic planning, to allow for control protocols that satisfy a set of constraints and minimize a cost function associated with the continuous or discrete states, as in the Programming-Based Optimized Mapping approach in Section 6.3. A few works have been recently published in this direction, which use optimization to generate control trajectories satisfying an LTL specification [204], or encode a reactive synthesis problem from STL specifications into an optimization problem executed in a receding horizon approach [171, 172].

- *Distributed protocols.* We can develop more systematic methods for designing and validating formal interface specifications between subsystems (horizontal contracts) that allow verification and synthesis to be performed at the subsystem level, with guaranteed system-level requirements. The problem of distributed control synthesis from temporal logic specifications by following a given controller architecture [163] and the one of control synthesis by using a library of pre-existing components [131] are undecidable in general. Then, an interesting direction will be to overcome both the limitations of the generic "architecture-based" and "library-based" approaches by adequately combining them. On the one hand, by using a reference component library as in Section 6.5, we limit the size of the exploration space and exploit built-in refinement assertions to speed up verification tasks. On the other hand, by using a reference control architecture, based on the plant architecture and the system-level requirements, we make the synthesis task tractable, by suggesting viable aggregations of contracts (e.g. realizable specifications) out of the contract library to build the controller in a bottom-up fashion. The efficient contract refinement checking algorithm proposed in this thesis showed indeed the benefits of combining a library-based approach, as in platform-based design, with contracts, to perform complex verification tasks in a hierarchical and modular way. A full-fledged theoretical study of the complexity of the proposed algorithm 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, e.g. starting with domain-related benchmarks, will also be object of future work. Further extensions of this work also include investigating algorithms for automatic mapping of library contracts to plant architectures, the adoption of learning algorithms for library optimization, and the definition of benchmarks and quality metrics to estimate the effectiveness of a library. We expect that approaches as the one in this dissertation, relying on design libraries to build proof systems that can incorporate design knowledge accumulated over the time, in terms of data collected both in the design and in the testing phases, will gain increasingly more momentum in the future, due to the recent advances of data storage and data analysis technologies.

- *Hierarchical control structures.* Hierarchical control structures can be developed that make use of demand-response architectures and formal interface specifications between layers (vertical contracts) to achieve a system-level goal. The preliminary work in both Section 6.3 and Section 6.5 will also provide a starting point for this effort. The hierarchical load management architecture for aircraft electrical power systems proposed in this thesis, together with the supporting design framework HOLMS [1], is novel and improves on the current state of the art. As future work, we intend to further characterize the proposed control scheme by formalizing the interaction between the high-level and low-level load management systems in terms of vertical contracts, to provide stronger guarantees of correctness and enable compositional development of the high-level and low-level controllers.

8.2.3 Applications

We illustrated our methodology and tools on the design of aircraft electrical and air management systems. However, the number of applications and future directions are endless, including, among others, mixed-signal integrated circuits, distributed and networked control systems, "smart" buildings [135], robotics, sensor and actuator "swarms" [9].

To better illustrate our methodology, we considered an abstract representation of a CPS in terms of composition between a controller and a plant. However, the concepts discussed in this thesis are general enough to encompass several other, if not all, categories of CPS. Specifically, because of the rigorous formalization of both the horizontal and vertical interactions between components, contracts seem to offer a "natural" theoretical framework for the design of provably correct *distributed* and *hierarchical control* systems in a scalable way, as discussed above. In this respect, to support the design of adaptive architectures, in which components (agents) can dynamically reconfigure themselves, e.g. by changing their locations or communication patterns, the challenge is to provide mechanisms that can efficiently export at the *architectural exploration level* the most important constraints and metrics imposed by the lower-level *system dynamics*, and *network fabrics*. Accordingly, as an integral part of the *execution platform refinement* process, which was not covered in this thesis, our framework can be extended to incorporate several design space exploration methodologies across the hardware, software and communication layers, which are being consolidated over the years by the joint effort of both academia and industry.

Arguably, we can further extend our design paradigm to systems whose main objective is to sense and monitor a physical "plant," and process the collected data, rather than controlling it. An example of such a system, still in the aircraft context, could be the built-in test equipment (BITE) of an electric power system, which collects data from a set of observation points (e.g., including sensors or software flags) to identify the state of the plant and take decisions about potential recovery actions for maintenance purposes. In this case, contracts would rather be used to capture the interaction of the physical world with the *sensing, identification, data analysis, or learning algorithms*, and their deployment on the embedded platform.

Bibliography

[1] "HOLMS: Optimal load management system for aircraft electric power distribution," https://github.com/forresti/optimal_load_management, accessed: 2015-07-27.

[2] *Modelica Language*. [Online]. Available: http://www.modelica.org

[3] *OMG Systems Modeling Language*. [Online]. Available: http://www.sysml.org/

[4] UPPAAL-Tiga, a synthesis tool for timed games. [Online]. Available: http://people.cs.aau.dk/~adavid/tiga/

[5] (2012, Feb.) IBM ILOG CPLEX Optimizer. [Online]. Available: www.ibm.com/software/integration/optimization/cplex-optimizer/

[6] M. Abadi and L. Cardelli, *A Theory of Objects*. Springer-Verlag, 1996.

[7] M. Agrawal and P. Thiagarajan, "The discrete time behavior of lazy linear hybrid automata," in *Hybrid Systems: Computation and Control*, ser. Lecture Notes in Computer Science, M. Morari and L. Thiele, Eds. Springer Berlin Heidelberg, 2005, vol. 3414, pp. 55–69. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-31954-2_4

[8] D. E. N. Agut, D. A. van Beek, and J. E. Rooda, "Syntax and semantics of the compositional interchange format for hybrid systems," *J. Log. Algebr. Program.*, vol. 82, no. 1, pp. 1–52, 2013.

[9] B. Aksanli, A. S. Akyurek, M. Behl, M. Clark, A. Donzé, P. Dutta, P. Lazik, M. Maasoumy, R. Mangharam, T. X. Nghiem, V. Raman, A. Rowe, A. Sangiovanni-Vincentelli, S. Seshia, T. S. Rosing, and J. Venkatesh, "Distributed control of a swarm of buildings connected to a smart grid: Demo abstract," in *Proceedings of the 1st ACM Conference on Embedded Systems for Energy-Efficient Buildings*, ser. BuildSys '14. New York, NY, USA: ACM, 2014, pp. 172–173. [Online]. Available: http://doi.acm.org/10.1145/2674061.2675019

[10] L. D. Alfaro and T. A. Henzinger, "Interface theories for component-based design," in *Proc. ACM IEEE Int. Conf. Embedded Software*. Springer-Verlag, 2001, pp. 148–165.

[11] R. Alur, C. Courcoubetis, T. A. Henzinger, and P. H. Ho, "Hybrid Automata: An Algorithmic Approach to the Specification and Verification of Hybrid Systems," in *Hybrid Systems*, ser. LNCS, vol. 736. Springer, 1993, pp. 209–229.

[12] R. Alur, T. Henzinger, G. Lafferriere, and G. Pappas, "Discrete abstractions of hybrid systems," *Proc. IEEE*, vol. 88, no. 7, pp. 971–984, July 2000.

[13] R. Alur, T. Dang, and F. Ivančić, "Counterexample-guided predicate abstraction of hybrid systems," *Theoretical Computer Science*, vol. 354, no. 2, pp. 250–271, 2006.

[14] R. Alur and D. L. Dill, "A theory of timed automata," *Theor. Comput. Sci.*, vol. 126, no. 2, pp. 183–235, 1994. [Online]. Available: http://dx.doi.org/10.1016/0304-3975(94)90010-8

[15] R. Alur, R. Grosu, Y. Hur, V. Kumar, and I. Lee, "Modular specification of hybrid systems in Charon," in *Hybrid Systems: Computation and Control*, ser. LNCS, vol. 1790. Springer, 2000, pp. 6–19.

[16] R. Alur and T. A. Henzinger, "A really temporal logic," in *Symposium on Foundations of Computer Science*, 1989, pp. 164–169.

[17] R. Alur and T. Henzinger, "Reactive modules," *Formal Methods in System Design*, vol. 15, no. 1, pp. 7–48, 1999. [Online]. Available: http://dx.doi.org/10.1023/A%3A1008739929481

[18] Y. Annpureddy, C. Liu, G. E. Fainekos, and S. Sankaranarayanan, "S-TaLiRo: A tool for temporal logic falsification for hybrid systems," in *Proc. Int. Conf. Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2011, pp. 254–257.

[19] E. Asarin, A. Donzé, O. Maler, and D. Nickovic, "Parametric identification of temporal properties," in *Proc. Int. Conf. Runtime Verification*, ser. Lecture Notes in Computer Science, vol. 7186. Springer, 2012.

[20] E. Asarin, O. Bournez, T. Dang, and O. Maler, "Approximate reachability analysis of piecewise-linear dynamical systems," in *Hybrid Systems: Computation and Control*, ser. LNCS. Springer Berlin Heidelberg, 2000, vol. 1790, pp. 20–31. [Online]. Available: http://dx.doi.org/10.1007/3-540-46430-1_6

[21] L. Babai and E. M. Luks, "Canonical labeling of graphs," in *Proc. of ACM Symp. on Theory of Computing*, ser. STOC '83, 1983, pp. 171–183.

[22] C. Baier and J.-P. Katoen, *Principles of Model Checking*. Massachussetts, USA: The MIT Press, 2008.

[23] N. Bajaj, P. Nuzzo, M. Masin, and A. L. Sangiovanni-Vincentelli, "Optimized selection of reliable and cost-effective cyber-physical system architectures," in *Proc. Design, Automation and Test in Europe*, Mar. 2015.

[24] F. Balarin, H. Hsieh, L. Lavagno, C. Passerone, A. L. Sangiovanni-Vincentelli, and Y. Watanabe, "Metropolis: an integrated electronic system design environment," *Computer*, vol. 36, no. 4, 2003.

[25] F. Balarin, A. Davare, M. D'Angelo, D. Densmore, T. Meyerowitz, R. Passerone, A. Pinto, A. Sangiovanni-Vincentelli, A. Simalatsar, Y. Watanabe, G. Yang, and Q. Zhu, "Platform-based design and frameworks: METROPOLIS and METRO II," in *Model-Based Design for Embedded Systems*, G. Nicolescu and P. J. Mosterman, Eds. Boca Raton, London, New York: CRC Press, Taylor and Francis Group, November 2009, ch. 10, p. 259.

[26] A. Balluchi, L. Benvenuti, T. Villa, H. Wong-Toi, and A. L. Sangiovanni-Vincentelli, "Controller synthesis for hybrid systems with a lower bound on event separation," *International Journal of Control*, vol. 76, no. 12, pp. 1171–1200, Aug. 2003.

[27] J. Bals, G. Hofer, A. Pfeiffer, and C. Schallert, "Virtual iron bird – A multidisciplinary modelling and simulation platform for new aircraft system architectures," in *German Aerospace Conference*, 2005.

[28] C. Barrett, R. Sebastiani, S. A. Seshia, and C. Tinelli, "Satisfiability modulo theories," in *Handbook of Satisfiability*, A. Biere, H. van Maaren, and T. Walsh, Eds. IOS Press, 2009, vol. 4, ch. 8.

[29] S. S. Bauer, A. David, R. Hennicker, K. G. Larsen, A. Legay *et al.*, "Moving from specifications to contracts in component-based design," in *Int. Conf. on Fundamental Approaches to Software Engineering*. Springer, 2012, pp. 43–58.

[30] G. Behrmann, A. Cougnard, A. David, E. Fleury, K. G. Larsen, and D. Lime, "UPPAAL-Tiga: Time for playing games!" in *Proc. Int. Conf. Comput.-Aided Verification*. Springer, 2007, pp. 121–125.

[31] G. Behrmann, A. David, K. G. Larsen, P. Pettersson, and W. Yi, "Developing UPPAAL over 15 years," *Softw., Pract. Exper.*, vol. 41, no. 2, pp. 133–142, 2011. [Online]. Available: http://dx.doi.org/10.1002/spe.1006

[32] M. Benerecetti, M. Faella, and S. Minopoli, "Automatic synthesis of switching controllers for linear hybrid systems: Safety control," *Theor. Comput. Sci.*, vol. 493, pp. 116–138, 2013. [Online]. Available: http://dx.doi.org/10.1016/j.tcs.2012.10.042

[33] A. Benveniste, B. Caillaud, A. Ferrari, L. Mangeruca, R. Passerone, and C. Sofronis, "Multiple viewpoint contract-based specification and design," in *Formal Methods for Components and Objects*. Springer-Verlag, 2008, pp. 200–225.

[34] A. Benveniste, B. Caillaud, D. Nickovic, R. Passerone, J.-B. Raclet, P. Reinkemeier *et al.*, "Contracts for System Design," INRIA, Rapport de recherche RR-8147, Nov. 2012.

[35] A. Benveniste, B. Caillaud, and R. Passerone, "Multi-Viewpoint State Machines for Rich Component Models," in *Model-Based Design of Heterogeneous Embedded Systems*. CRC Press, 2009.

[36] L. Benvenuti, A. Ferrari, E. Mazzi, and A. Sangiovanni-Vincentelli, "Contract-based design for computation and verification of a closed-loop hybrid system," in *Proc. Hybrid Systems: Computation and Control*, 2008, pp. 58–71.

[37] L. Benvenuti, D. Bresolin, P. Collins, A. Ferrari, L. Geretti, and T. Villa, "Ariadne: Dominance checking of nonlinear hybrid automata using reachability analysis," in *Reachability Problems*, ser. Lecture Notes in Computer Science, A. Finkel, J. Leroux, and I. Potapov, Eds. Springer Berlin Heidelberg, 2012, vol. 7550, pp. 79–91. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-33512-9_8

[38] ——, "Assume-guarantee verification of nonlinear hybrid systems with ARIADNE," *Int. J. Robust Nonlinear Control*, vol. 24, no. 4, pp. 699–724, 2014.

[39] B. Bérard and L. Sierra, "Comparing verification with HyTech, KRONOS and Uppaal on the railroad crossing example," CNRS & ENS de Chachan, France, Tech. Rep. LSV-00-2, 2000.

[40] T. Bienmüller, W. Damm, and H. Wittke, "The Statemate verification environment," in *Proc. Int. Conf. Comput.-Aided Verification*, ser. Lecture Notes in Computer Science, E. A. Emerson and A. P. Sistla, Eds. Springer Berlin Heidelberg, 2000, vol. 1855, pp. 561–567. [Online]. Available: http://dx.doi.org/10.1007/10722167_45

[41] T. Blochwitz, M. Otter, J. Akesson, M. Arnold, C. Clauss, H. Elmqvist, M. Friedrich, A. Junghanns, J. Mauss, D. Neumerkel, H. Olsson, and A. Viel, "Functional mockup interface 2.0: The standard for tool independent exchange of simulation models," 2012.

[42] R. Bloem, K. Chatterjee, K. Greimel, T. Henzinger, G. Hofferek, B. Jobstmann, B. Knighofer, and R. Knighofer, "Synthesizing robust systems," *Acta Informatica*, vol. 51, no. 3-4, pp. 193–220, 2014. [Online]. Available: http://dx.doi.org/10.1007/s00236-013-0191-5

[43] R. Bloem, A. Cimatti, K. Greimel, G. Hofferek, R. Knighofer, M. Roveri, V. Schuppan, and R. Seeber, "RATSY – a new requirements analysis tool with synthesis," in *Proc. Int. Conf. Comput.-Aided Verification*, ser. Lecture Notes in Computer Science, T. Touili, B. Cook, and P. Jackson, Eds. Springer Berlin Heidelberg, 2010, vol. 6174, pp. 425–429. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-14295-6_37

[44] A. Bohy, V. Bruyére, E. Filiot, N. Jin, and J.-F. Raskin, "Acacia+: a tool for LTL synthesis," in *Proc. Int. Conf. Comput.-Aided Verification*, ser. CAV'12. Berlin, Heidelberg: Springer-Verlag, 2012, pp. 652–657. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-31424-7_45

[45] G. Booch, J. Rumbaugh, and I. Jacobson, *The Unified Modeling Language User Guide (Addison-Wesley Object Technology Series)*, 2nd ed., Addison-Wesley Professional, San Jose, USA, 2005.

[46] D. Bresolin, L. Di Guglielmo, L. Geretti, R. Muradore, P. Fiorini, and T. Villa, "Open problems in verification and refinement of autonomous robotic systems," in *Euromicro Conference on Digital System Design*, Sept 2012, pp. 469–476.

[47] D. Bresolin, L. Di Guglielmo, L. Geretti, and T. Villa, "Correct-by-construction code generation from hybrid automata specification," in *International Wireless Communications and Mobile Computing Conference*, July 2011, pp. 1660–1665.

[48] D. Broman, C. Brooks, L. Greenberg, E. A. Lee, M. Masin, S. Tripakis, and M. Wetter, "Determinate composition of FMUs for co-simulation," in *Proc. ACM IEEE Int. Conf. Embedded Software*. Piscataway, NJ, USA: IEEE Press, 2013, pp. 2:1–2:12. [Online]. Available: http://dl.acm.org/citation.cfm?id=2555754.2555756

[49] B. Caillaud, B. Delahaye, K. Larsen, A. Legay, M. Pedersen, and A. Wasowski, "Compositional design methodology with Constraint Markov Chains," in *International Conference on the Quantitative Evaluation of Systems*, Sept 2010, pp. 123–132.

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

[51] A. Casagrande and T. Dreossi, "pyHybrid analysis: A package for semantics analysis of hybrid systems," in *Euromicro Conf. Digital System Design*, Sep. 2013, pp. 815–818.

[52] A. Casagrande, C. Piazza, and A. Policriti, "Discrete semantics for hybrid automata," *Discrete Event Dynamic Systems*, vol. 19, no. 4, pp. 471–493, Dec. 2009. [Online]. Available: http://dx.doi.org/10.1007/s10626-009-0082-7

[53] C. Cassandras and S. Lafortune, *Introduction to Discrete Event Systems*, ser. Springer-Link Engineering. Springer, 2008.

[54] S. Chandrasekaran, S. Ragon, D. Lindner, Z. Gurdal, and D. Boroyevich, "Optimization of an aircraft power distribution subsystem," *Journal of aircraft*, vol. 40, no. 1, pp. 16–26, 2003.

[55] K. Chaudhuri, D. Doligez, L. Lamport, and S. Merz, "The TLA+ proof system: Building a heterogeneous verification platform," in *Proc. Int. Colloquium Conf. Theoretical Aspects of Computing*, ser. ICTAC'10. Berlin, Heidelberg: Springer-Verlag, 2010, pp. 44–44. [Online]. Available: http://dl.acm.org/citation.cfm?id=1881833.1881837

[56] X. Chen, E. Ábrahám, and S. Sankaranarayanan, "Flow*: An analyzer for non-linear hybrid systems," in *Proc. Int. Conf. Comput.-Aided Verification*, ser. Lecture Notes in Computer Science, vol. 8044. Springer Berlin Heidelberg, 2013, pp. 258–263.

[57] A. Chutinan and B. Krogh, "Verification of infinite-state dynamic systems using approximate quotient transition systems," *IEEE Trans. Automatic Control*, vol. 46, no. 9, pp. 1401–1410, Sep 2001.

[58] A. Cimatti and S. Tonetta, "A property-based proof system for contract-based design," in *EUROMICRO Conference on Software Engineering and Advanced Applications*, 2012, pp. 21–28.

[59] A. Cimatti, E. M. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella, "NuSMV 2: An open source tool for symbolic model checking," in *Proc. Int. Conf. Comput.-Aided Verification*, 2002.

[60] A. Cimatti, S. Mover, and S. Tonetta, "SMT-based scenario verification for hybrid systems," *Formal Methods in System Design*, vol. 42, no. 1, pp. 46–66, 2013. [Online]. Available: http://dx.doi.org/10.1007/s10703-012-0158-0

[61] A. Cimatti, M. Roveri, and S. Tonetta, "Requirements validation for hybrid systems," in *Computer Aided Verification*, ser. Lecture Notes in Computer Science, A. Bouajjani and O. Maler, Eds. Springer Berlin Heidelberg, 2009, vol. 5643, pp. 188–203.

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

[63] E. M. Clarke, A. Fehnker, Z. Han, B. H. Krogh, J. Ouaknine, O. Stursberg, and M. Theobald, "Abstraction and counterexample-guided refinement in model checking of hybrid systems," *Int. J. Found. Comput. Sci.*, vol. 14, no. 4, pp. 583–604, 2003. [Online]. Available: http://dx.doi.org/10.1142/S012905410300190X

[64] J. Cobleigh, D. Giannakopoulou, and C. Pasareanu, "Learning assumptions for compositional verification," in *Tools and Algorithms for the Construction and Analysis of Systems*, 2003, pp. 331–346.

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

[66] W. Damm, G. Pinto, and S. Ratschan, "Guaranteed termination in the verification of LTL properties of non-linear robust discrete time hybrid systems," *Int. J. Foundations of Computer Science*, vol. 18, no. 01, pp. 63–86, 2007. [Online]. Available: http://www.worldscientific.com/doi/abs/10.1142/S0129054107004577

[67] W. Damm, A. Votintseva, A. Metzner, B. Josko, T. Peikenkamp, and E. Böde, "Boosting re-use of embedded automotive applications through rich components," *Proc. Foundations of Interface Technologies*, 2005.

[68] T. Dang, O. Maler, and R. Testylier, "Accurate hybridization of nonlinear systems," in *Proc. Hybrid Systems: Computation and Control*, ser. HSCC '10. New York, NY, USA: ACM, 2010, pp. 11–20. [Online]. Available: http://doi.acm.org/10.1145/1755952.1755956

[69] L. de Alfaro and T. Henzinger, "Interface theories for component-based design," in *Proc. Int. Conf. Embedded Software*. Springer, LNCS 2211, 2001.

[70] L. de Alfaro and T. A. Henzinger, "Interface automata," in *Proc. Symp. Foundations of Software Engineering*. ACM Press, 2001, pp. 109–120.

[71] M. De Wulf, L. Doyen, and J.-F. Raskin, "Almost ASAP semantics: From timed models to timed implementations," in *Hybrid Systems: Computation and Control*, ser. Lecture Notes in Computer Science, R. Alur and G. Pappas, Eds. Springer Berlin Heidelberg, 2004, vol. 2993, pp. 296–310. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-24743-2_20

[72] P. Derler, E. A. Lee, and A. Sangiovanni-Vincentelli, "Modeling cyber-physical systems," *Proc. IEEE*, vol. 100, no. 1, pp. 13–28, January 2012.

[73] P. Derler, E. A. Lee, S. Tripakis, and M. Törngren, "Cyber-physical system design contracts," in *Proc. Int. Conf. Cyber-Physical Systems*, 2013, pp. 109–118. [Online]. Available: http://doi.acm.org/10.1145/2502524.2502540

[74] S. Di Cairano, A. Bemporad, M. Kvasnica, and M. Morari, "An architecture for data interchange of switched linear systems," HYCON Network of Excellence, Deliverable workpackage 3.D, 2006.

[75] L. Di Guglielmo, S. A. Seshia, and T. Villa, "Synthesis of implementable control strategies for lazy linear hybrid automata," in *Federated Conference on Computer Science and Information Systems*, Sept 2013, pp. 1381–1388.

[76] A. Donzé and O. Maler, "Robust satisfaction of temporal logic over real-valued signals," in *Formal Modeling and Analysis of Timed Systems*, 2010, pp. 92–106.

[77] A. Donzé, "Breach, a toolbox for verification and parameter synthesis of hybrid systems," in *Proc. Int. Conf. Comput.-Aided Verification*. Berlin, Heidelberg: Springer-Verlag, 2010, pp. 167–170.

[78] L. Doyen, T. A. Henzinger, B. Jobstmann, and T. Petrov, "Interface theories with component reuse," in *Proc. ACM IEEE Int. Conf. Embedded Software*, 2008, pp. 79–88.

[79] E. A. Emerson, "Temporal and modal logic," *Handbook of theoretical computer science*, vol. 2, pp. 995–1072, 1990.

[80] J. Finn, P. Nuzzo, and A. Sangiovanni-Vincentelli, "A mixed discrete-continuous optimization scheme for cyber-physical system architecture exploration," in *Proc. IEEE/ACM Int. Conf. Comput.-Aided Design*, Nov. 2015, to appear.

[81] G. Frehse, "PHAVer: algorithmic verification of hybrid systems past HyTech," *Int. J. Software Tools for Technology Transfer*, vol. 10, pp. 263–279, 2008.

[82] G. Frehse, C. Le Guernic, A. Donzé, S. Cotton, R. Ray, O. Lebeltel, R. Ripado, A. Girard, T. Dang, and O. Maler, "SpaceEx: Scalable verification of hybrid systems," in *Proc. Int. Conf. Comput.-Aided Verification*, ser. LNCS. Springer Berlin / Heidelberg, 2011, vol. 6806, pp. 379–395.

[83] T. French and M. Reynolds, "A sound and complete proof system for QPTL," in *Advances in Modal Logic*. King's College Publications, 2003, vol. 4, pp. 127–148.

[84] C. E. Garcia, D. M. Prett, and M. Morari, "Model predictive control: Theory and practice – A survey," *Automatica*, vol. 25, no. 3, pp. 335–348, 1989. [Online]. Available: http://www.sciencedirect.com/science/article/pii/0005109889900022

[85] A. Ghosal, A. Sangiovanni-Vincentelli, C. M. Kirsch, T. A. Henzinger, and D. Iercan, "A hierarchical coordination language for interacting real-time tasks," in *Proc. ACM IEEE Int. Conf. Embedded Software*. New York, NY, USA: ACM, 2006, pp. 132–141. [Online]. Available: http://doi.acm.org/10.1145/1176887.1176907

[86] A. Girard, G. Pola, and P. Tabuada, "Approximately bisimilar symbolic models for incrementally stable switched systems," *IEEE Transactions on Automatic Control*, vol. 55, no. 1, pp. 116–126, Jan 2010.

[87] G. Gössler, D. N. Xu, and A. Girault, "Probabilistic contracts for component-based design," *Formal Methods in System Design*, vol. 41, no. 2, pp. 211–231, 2012. [Online]. Available: http://dx.doi.org/10.1007/s10703-012-0162-4

[88] S. Graf, R. Passerone, and S. Quinton, "Contract-based reasoning for component systems with rich interactions," in *Embedded Systems Development*, ser. Embedded Systems, A. Sangiovanni-Vincentelli, H. Zeng, M. Di Natale, and P. Marwedel, Eds. Springer New York, 2014, vol. 20, pp. 139–154. [Online]. Available: http://dx.doi.org/10.1007/978-1-4614-3879-3_8

[89] O. Grumberg and D. E. Long, "Model checking and modular verification," *ACM Transactions on Programming Languages and Systems*, vol. 16, 1991.

[90] L. Guo, Z. Qi, P. Nuzzo, R. Passerone, A. Sangiovanni-Vincentelli, and E. A. Lee, "Metronomy: A function-architecture co-simulation framework for timing verification of cyber-physical systems," in *Proc. Int. Conf. Hardware-Software Codesign and System Synthesis*, Oct. 2014.

[91] A. Gupta, K. McMillan, and Z. Fu, "Automated assumption generation for compositional verification," *Formal Methods in System Design*, pp. 285–301, 2008.

[92] C. Hang, P. Manolios, and V. Papavasileiou, "Synthesizing cyber-physical architectural models with real-time constraints," in *Proc. Int. Conf. Comput.-Aided Verification*, Dec. 2011.

[93] P. Helle, M. Masin, and L. Greenberg, "Approximate reliability algebra for architecture optimization," in *Proc. Int. Conf. on Computer Safety, Reliability, and Security*, 2012, pp. 279–290.

[94] T. A. Henzinger, "The theory of hybrid automata," in *Proc. IEEE Symp. Logic in Computer Science*, Jul. 1996, pp. 278–292.

[95] T. A. Henzinger and D. Nickovic, "Independent implementability of viewpoints," in *Monterey Workshop*. Springer, 2012, pp. 380–395.

[96] T. Henzinger, P.-H. Ho, and H. Wong-Toi, "Algorithmic analysis of nonlinear hybrid systems," *IEEE Trans. Automatic Control*, vol. 43, no. 4, pp. 540–554, Apr. 1998.

[97] T. Henzinger, B. Horowitz, and C. Kirsch, "Giotto: a time-triggered language for embedded programming," *Proc. IEEE*, vol. 91, no. 1, pp. 84–99, Jan 2003.

[98] T. A. Henzinger, P. Ho, and H. Wong-Toi, "HYTECH: A model checker for hybrid systems," *Int. J. Software Tools for Technology Transfer*, vol. 1, no. 1-2, pp. 110–122, 1997.

[99] T. A. Henzinger, P. W. Kopke, A. Puri, and P. Varaiya, "What's decidable about hybrid automata?" *J. Comput. Syst. Sci.*, vol. 57, no. 1, pp. 94–124, 1998. [Online]. Available: http://dx.doi.org/10.1006/jcss.1998.1581

[100] A. Iannopollo, P. Nuzzo, S. Tripakis, and A. L. Sangiovanni-Vincentelli, "Library-based scalable refinement checking for contract-based design," in *Proc. Design, Automation and Test in Europe*, Mar. 2014.

[101] S. Jha, B. A. Brady, and S. A. Seshia, "Symbolic reachability analysis of lazy linear hybrid automata," in *Proc. 5th International Conference on Formal Modeling and Analysis of Timed Systems (FORMATS)*, ser. Lecture Notes in Computer Science, vol. 4763, October 2007, pp. 241–256.

[102] S. Jha, S. Gulwani, S. A. Seshia, and A. Tiwari, "Synthesizing switching logic for safety and dwell-time requirements," in *Proceedings of the International Conference on Cyber-Physical Systems (ICCPS)*, April 2010, pp. 22–31.

[103] S. Jha, S. A. Seshia, and A. Tiwari, "Synthesis of optimal switching logic for hybrid systems," in *Proceedings of the International Conference on Embedded Software (EMSOFT)*, October 2011, pp. 107–116.

[104] S. K. Jha, "Towards automated system synthesis using SCIDUCTION," Ph.D. dissertation, EECS Department, University of California, Berkeley, Nov 2011. [Online]. Available: http://www.eecs.berkeley.edu/Pubs/TechRpts/2011/EECS-2011-118.html

[105] X. Jin, A. Donzé, J. Deshmukh, and S. A. Seshia, "Mining requirements from closed-loop control models," in *Proceedings of the International Conference on Hybrid Systems: Computation and Control (HSCC)*, April 2013.

[106] B. Jobstmann, S. Galler, M. Weiglhofer, and R. Bloem, "Anzu: A tool for property synthesis," in *Proc. Int. Conf. Comput.-Aided Verification*, ser. Lecture Notes in Computer Science, W. Damm and H. Hermanns, Eds. Springer Berlin Heidelberg, 2007, vol. 4590, pp. 258–262. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-73368-3_29

[107] T. Jomier *et al.*, "Final MOET technical report," Tech. Rep., Dec. 2009. [Online]. Available: http://www.eurtd.com/moet/

[108] B. Kaiser, P. Liggesmeyer, and O. Mäckel, "A new component concept for fault trees," in *Proc. Australian Workshop on Safety Critical Systems and Software*, 2003.

[109] Y. Kesten and A. Pnueli, "A complete proof systems for QPTL," in *Proc. IEEE Symp. on Logic in Computer Science*, Jun 1995, pp. 2–12.

[110] K. Keutzer, S. Malik, R. Newton, J. Rabaey, and A. Sangiovanni Vincentelli, "System Level Design: Orthogonalization of Concerns and Platform-Based Design," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 19, no. 12, pp. 1523–1543, 2000.

[111] M. Kloetzer and C. Belta, "A fully automated framework for control of linear systems from temporal logic specifications," *IEEE Trans. Autom. Control*, vol. 53, no. 1, pp. 287–297, Feb. 2008.

[112] R. Koymans, "Specifying real-time properties with metric temporal logic," *Real-Time Syst.*, vol. 2, no. 4, pp. 255–299, 1990.

[113] H. Kress-Gazit, G. Fainekos, and G. Pappas, "Temporal-logic-based reactive mission and motion planning," *IEEE Trans. Robot.*, vol. 25, no. 6, pp. 1370–1381, Dec 2009.

[114] P. Krus and J. Nyman, "Complete aircraft system simulation for aircraft design – Paradigms for modelling of complex systems," in *Int. Congress of Aeronautical Sciences*, 2000.

[115] O. Kupferman and M. Y. Vardi, "Vacuity detection in temporal model checking," *Int. J. Software Tools for Technology Transfer*, vol. 4, no. 2, pp. 224–233, 2003.

[116] T. Kurtoglu, P. Bunus, and J. de Kleer, "Simulation-based design of aircraft electrical power systems," in *Int. Modelica Conf.*, Mar. 2011, pp. 92–106.

[117] M. Kwiatkowska, G. Norman, and D. Parker, "Stochastic model checking," in *Formal Methods for the Design of Computer, Communication and Software Systems: Performance Evaluation (SFM'07)*, ser. LNCS (Tutorial Volume), M. Bernardo and J. Hillston, Eds., vol. 4486. Springer, 2007, pp. 220–270.

[118] ——, "PRISM 4.0: Verification of probabilistic real-time systems," in *Proc. Int. Conf. Comput.-Aided Verification*, ser. LNCS, G. Gopalakrishnan and S. Qadeer, Eds., vol. 6806. Springer, 2011, pp. 585–591.

[119] M. Kwiatkowska, G. Norman, D. Parker, and H. Qu, "Assume-guarantee verification for probabilistic systems," in *Proc. Int. Conf. Tools and Algorithms for the Construction and Analysis of Systems*, ser. Lecture Notes in Computer Science, J. Esparza and R. Majumdar, Eds. Springer Berlin Heidelberg, 2010, vol. 6015, pp. 23–37. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-12002-2_3

[120] E. A. Lee, "Cyber physical systems: Design challenges," in *Proc. IEEE Int. Symposium on Object Oriented Real-Time Distributed Computing*, May 2008, pp. 363–369.

[121] E. A. Lee and A. Sangiovanni-Vincentelli, "A framework for comparing models of computation," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 17, no. 12, pp. 1217–1229, Dec. 1998.

[122] E. A. Lee, "The problem with threads," *Computer*, vol. 39, no. 5, pp. 33–42, May 2006. [Online]. Available: http://dx.doi.org/10.1109/MC.2006.180

[123] E. A. Lee and S. A. Seshia, *Introduction to Embedded Systems, A Cyber-Physical Systems Approach*. http://LeeSeshia.org, 2014, Edition 1.5.

[124] ——, *Introduction to Embedded Systems, A Cyber-Physical Systems Approach*, Second ed. http://LeeSeshia.org, 2015.

[125] M.-K. Leung, T. Mandl, E. Lee, E. Latronico, C. Shelton, S. Tripakis, and B. Lickly, "Scalable semantic annotation using lattice-based ontologies," in *Model Driven Engineering Languages and Systems*, ser. Lecture Notes in Computer Science, A. Schürr and B. Selic, Eds. Springer Berlin Heidelberg, 2009, vol. 5795, pp. 393–407. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-04425-0_31

[126] W. Li, L. Dworkin, and S. A. Seshia, "Mining assumptions for synthesis," in *Int. Conf. Formal Methods and Models for Co-Design*, July 2011, pp. 43–50.

[127] W. Li, A. Forin, and S. A. Seshia, "Scalable specification mining for verification and diagnosis," in *Proc. IEEE/ACM Design Automation Conf.* New York, NY, USA: ACM, 2010, pp. 755–760.

[128] W. Li, D. Sadigh, S. S. Sastry, and S. A. Seshia, "Synthesis for human-in-the-loop control systems," in *Proceedings of the 20th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, April 2014, pp. 470–484.

[129] J. Löfberg, "YALMIP: A toolbox for modeling and optimization in MATLAB," in *Int. Symp. Computer Aided Control Systems Design*, 2004, pp. 284–289.

[130] C. Lucet and J.-F. Manouvrier, "Exact methods to compute network reliability," in *Proc. Int. Conf. on Mathematical Methods in Reliability*, 1997.

[131] Y. Lustig and M. Vardi, "Synthesis from component libraries," in *Foundations of Software Science and Computational Structures*, ser. Lecture Notes in Computer Science, L. de Alfaro, Ed. Springer Berlin Heidelberg, 2009, vol. 5504, pp. 395–409. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-00596-1_28

[132] N. Lynch, R. Segala, and F. Vaandrager, "Hybrid I/O automata," *Information and Computation*, vol. 185, no. 1, pp. 105 – 157, 2003.

[133] M. R. Lyu *et al.*, *Handbook of software reliability engineering.* IEEE Computer Society Press CA, 1996, vol. 3.

[134] M. Maasoumy, P. Nuzzo, F. Iandola, M. Kamgarpour, A. Sangiovanni-Vincentelli, and C. Tomlin, "Optimal load management system for aircraft electric power distribution," in *Int. Conf. Decision and Control*, Dec 2013, pp. 2939–2945.

[135] M. Maasoumy, P. Nuzzo, and A. Sangiovanni-Vincentelli, "Smart buildings in the smart grid: Contract-based design of an integrated energy management system," in *Cyber Physical Systems Approach to Smart Electric Power Grid*, ser. Power Systems, S. K. Khaitan, J. D. McCalley, and C. C. Liu, Eds. Springer Berlin Heidelberg, 2015, pp. 103–132. [Online]. Available: http://dx.doi.org/10.1007/978-3-662-45928-7_5

[136] O. Maler and D. Nickovic, "Monitoring temporal properties of continuous signals," in *Formal Modeling and Analysis of Timed Systems*, 2004, pp. 152–166.

[137] O. Maler, A. Pnueli, and J. Sifakis, "On the synthesis of discrete controllers for timed systems (an extended abstract)," in *STACS*, 1995, pp. 229–242. [Online]. Available: http://dx.doi.org/10.1007/3-540-59042-0_76

[138] T. Mancini, F. Mari, A. Massini, I. Melatti, F. Merli, and E. Tronci, "System level formal verification via model checking driven simulation," in *Proc. Int. Conf. Comput.-Aided Verification*, ser. Lecture Notes in Computer Science, vol. 8044. Springer - Verlag, 2013, pp. 296–312.

[139] Z. Manna and A. Pnueli, *The temporal logic of reactive and concurrent systems: Specification*. Springer, 1992, vol. 1.

[140] F. Mari, I. Melatti, I. Salvo, and E. Tronci, "Model based synthesis of control software from system level formal specifications," *ACM Transactions on Software Engineering and Methodology*, vol. 23, no. 1, 2014, Article 6.

[141] M. Masin, A. Sangiovanni-Vincentelli, A. Ferrari, L. Mangeruca, H. Broodney, L. Greenberg, M. Sambur, D. Dotan, S. Zolotnizky, and S. Zadorozhniy, "META II: Lingua franca design and integration language," Tech. Rep., Aug. 2011. [Online]. Available: http://www.darpa.mil/uploadedFiles/Content/Our_Work/TTO/Programs/AVM/IBM META Final Report.pdf

[142] M. Mazo Jr., A. Davitian, and P. Tabuada, "PESSOA: A tool for embedded controller synthesis," in *Computer Aided Verification*, ser. LNCS, vol. 6174, 2010, pp. 566–569. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-14295-6_49

[143] B. Meyer, "Applying "design by contract"," *Computer*, vol. 25, no. 10, pp. 40–51, Oct. 1992.

[144] R. G. Michalko, "Electrical starting, generation, conversion and distribution system architecture for a more electric vehicle," *US Patent 7,439,634 B2*, Oct. 2008.

[145] MoBIES team, "HSIF semantics," University of Pennsylvania, Tech. Rep., 2002.

[146] MODELISAR Consortium and Modelica Association, *Functional Mock-up Interface for Co-Simulation. Version 1.0*. Retrieved from https://www.fmi- standard.org, Oct. 2010.

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

[148] P. Nuzzo, J. Finn, A. Iannopollo, and A. L. Sangiovanni-Vincentelli, "Contract-based design of control protocols for safety-critical cyber-physical systems," in *Proc. Design, Automation and Test in Europe*, Mar. 2014, pp. 1–4.

[149] P. Nuzzo, A. Iannopollo, S. Tripakis, and A. L. Sangiovanni-Vincentelli, "Are interface theories equivalent to contract theories?" in *Int. Conf. Formal Methods and Models for Co-Design*, Oct. 2014.

[150] P. Nuzzo, A. Puggelli, S. Seshia, and A. Sangiovanni-Vincentelli, "CalCS: SMT solving for non-linear convex constraints," in *Proc. Formal Methods in Computer-Aided Design*, Oct. 2010, pp. 71–79.

[151] P. Nuzzo, A. Sangiovanni-Vincentelli, D. Bresolin, L. Geretti, and T. Villa, "A platform-based design methodology with contracts and related tools for the design of cyber-physical systems," *Proc. IEEE*, to appear, 2015.

[152] P. Nuzzo, A. Sangiovanni-Vincentelli, X. Sun, and A. Puggelli, "Methodology for the design of analog integrated interfaces using contracts," *IEEE Sensors J.*, vol. 12, no. 12, pp. 3329–3345, Dec. 2012.

[153] P. Nuzzo, A. L. Sangiovanni-Vincentelli, and R. M. Murray, "Methodology and tools for next generation cyber-physical systems: The iCyPhy approach," in *Proc. INCOSE Int. Symp.*, Jul. 2015.

[154] P. Nuzzo, H. Xu, N. Ozay, J. Finn, A. Sangiovanni-Vincentelli, R. Murray, A. Donzé, and S. Seshia, "A contract-based methodology for aircraft electric power system design," *IEEE Access*, vol. 2, pp. 1–25, 2014.

[155] P. Nuzzo and A. Sangiovanni-Vincentelli, "Robustness in analog systems: Design techniques, methodologies and tools," in *Proc. IEEE Symp. Industrial Embedded Systems*, Jun. 2011.

[156] ——, "Let's get physical: Computer science meets systems," in *From Programs to Systems. The Systems perspective in Computing*, ser. Lecture Notes in Computer Science, S. Bensalem, Y. Lakhneck, and A. Legay, Eds. Springer Berlin Heidelberg, 2014, vol. 8415, pp. 193–208. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-54848-2_13

[157] N. Ozay, U. Topcu, and R. M. Murray, "Distributed power allocation for vehicle management systems," in *Int. Conf. Decision and Control*, 2011, pp. 4841–4848.

[158] A. Pinto, S. Becz, and H. M. Reeve, "Correct-by-construction design of aircraft electric power systems," in *AIAA Aviation Technology, Integration, and Operations Conf.*, 2010.

[159] A. Pinto, L. P. Carloni, R. Passerone, and A. L. Sangiovanni-Vincentelli, "Interchange format for hybrid systems: Abstract semantics," in *International Workshop on Hybrid Systems: Computation and Control*. Springer, 2006, pp. 491–506.

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

[161] A. Platzer, *Logical Analysis of Hybrid Systems: Proving Theorems for Complex Dynamics*. Heidelberg: Springer, 2010.

[162] A. Pnueli and R. Rosner, "On the synthesis of a reactive module," in *Proc. Symp. Principles of Programming Languages*. New York, NY, USA: ACM, 1989, pp. 179–190.

[163] ——, "Distributed reactive systems are hard to synthesize," in *Proc. Annual Symp. on Foundations of Computer Science*, vol. 2, Oct. 1990, pp. 746–757.

[164] A. Pnueli, "The temporal logic of programs," in *Annual Symp. on Foundations of Computer Science*, Nov. 1977, pp. 46–57.

[165] A. Pnueli, Y. Sa'ar, and L. D. Zuck, "Jtlv: A framework for developing verification algorithms," in *Computer Aided Verification*, ser. Lecture Notes in Computer Science, T. Touili, B. Cook, and P. Jackson, Eds. Springer Berlin Heidelberg, 2010, vol. 6174, pp. 171–174.

[166] V. Preoteasa and S. Tripakis, "Refinement Calculus of Reactive Systems," in *Proc. ACM IEEE Int. Conf. Embedded Software*, Oct. 2014.

[167] S. Quinton, S. Graf, and R. Passerone, "Contract-based reasoning for component systems with complex interactions," Verimag Research Report, Tech. Rep. TR-2010-12, 2010.

[168] J.-B. Raclet, E. Badouel, A. Benveniste, B. Caillaud, A. Legay, and R. Passerone, "Modal interfaces: unifying interface automata and modal specifications," in *Proc. ACM IEEE Int. Conf. Embedded Software*. New York, NY, USA: ACM, 2009, pp. 87–96. [Online]. Available: http://doi.acm.org/10.1145/1629335.1629348

[169] A. Rajhans, A. Bhave, I. Ruchkin, B. H. Krogh, D. Garlan, A. Platzer, and B. Schmerl, "Supporting heterogeneity in cyber-physical systems architectures," *IEEE Trans. Automatic Control*, vol. 59, no. 12, pp. 3178–3193, Dec 2014.

[170] P. Ramadge and W. Wonham, "The control of discrete event systems," *Proc. IEEE*, vol. 77, no. 1, pp. 81–98, Jan 1989.

[171] V. Raman, A. Donze, M. Maasoumy, R. M. Murray, A. Sangiovanni-Vincentelli, and S. A. Seshia, "Model predictive control with signal temporal logic specifications," in *Int. Conf. Decision and Control*, Dec 2014.

[172] V. Raman, A. Donzé, D. Sadigh, R. M. Murray, and S. A. Seshia, "Reactive synthesis from signal temporal logic specifications," in *Proc. Hybrid Systems: Computation and Control*, ser. HSCC '15. New York, NY, USA: ACM, 2015, pp. 239–248. [Online]. Available: http://doi.acm.org/10.1145/2728606.2728628

[173] S. Ratschan and Z. She, "Safety verification of hybrid systems by constraint propagation based abstraction refinement," *ACM Transactions in Embedded Computing Systems*, vol. 6, no. 1, 2007.

[174] I. Saha, R. Ramaithitima, V. Kumar, G. J. Pappas, and S. A. Seshia, "Automated composition of motion primitives for multi-robot systems from safe LTL specifications," in *Proc. IEEE/RSJ Int. Conf. Intelligent Robots and Systems (IROS)*, Sep. 2014.

[175] K. Sampigethaya and R. Poovendran, "Aviation cyber-physical systems: Foundations for future aircraft and air transport," *Proc. IEEE*, vol. 101, no. 8, pp. 1834–1855, 2013.

[176] A. Sangiovanni-Vincentelli, "Quo vadis, SLD? Reasoning about the trends and challenges of system level design," *Proc. IEEE*, vol. 95, no. 3, pp. 467–506, Mar. 2007.

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

[178] D. C. Schmidt, "Guest editor's introduction: Model-driven engineering," *Computer*, vol. 39, no. 2, pp. 25–31, Feb. 2006.

[179] B. Selic, "The pragmatics of model-driven development," *IEEE Software*, vol. 20, no. 5, pp. 19–25, 2003.

[180] S. A. Seshia, "Adaptive eager boolean encoding for arithmetic reasoning in verification," Ph.D. dissertation, Carnegie Mellon University, May 2005.

[181] ——, "Sciduction: Combining induction, deduction, and structure for verification and synthesis," in *Proceedings of the Design Automation Conference (DAC)*, June 2012, pp. 356–365.

[182] S. A. Seshia and A. Rakhlin, "Game-theoretic timing analysis," in *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE Press, November 2008, pp. 575–582.

[183] ——, "Quantitative analysis of systems using game-theoretic learning," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 11, no. S2, pp. 55:1–55:27, 2012.

[184] A. A. Shah, D. Schaefer, and C. J. J. Paredis, "Enabling multi-view modeling with SysML profiles and model transformations," in *Proc. Int. Conf. Product Lifecycle Management*, 2009.

[185] Y. Shoukry, P. Nuzzo, A. Puggelli, A. L. Sangiovanni-Vincentelli, S. A. Seshia, M. Srivastava, and P. Tabuada, "IMHOTEP-SMT: a Satisfiability Modulo Theory solver for secure state estimation," in *Proc. Int. Workshop on Satisfiability Modulo Theories*, 2015.

[186] Y. Shoukry, P. Nuzzo, A. Puggelli, A. L. Sangiovanni-Vincentelli, S. A. Seshia, and P. Tabuada, "Secure State Estimation Under Sensor Attacks: A Satisfiability Modulo Theory Approach," *ArXiv e-prints*, Dec. 2014, [online] http://adsabs.harvard.edu/abs/2014arXiv1412.4324S.

[187] Y. Shoukry, A. Puggelli, P. Nuzzo, A. L. Sangiovanni-Vincentelli, S. A. Seshia, and P. Tabuada, "Sound and complete state estimation for linear dynamical systems under sensor attack using satisfiability modulo theory solving," in *Proc. IEEE American Control Conference*, 2015.

[188] B. I. Silva, K. Richeson, B. H. Krogh, and A. Chutinan, "Modeling and verification of hybrid dynamical system using CheckMate," in *ADPM*, 2000.

[189] A. P. Sistla and E. M. Clarke, "The complexity of propositional linear temporal logics," *J. ACM*, vol. 32, no. 3, pp. 733–749, 1985.

[190] A. P. Sistla, M. Y. Vardi, and P. Wolper, "The complementation problem for Büchi automata with applications to temporal logic," in *Automata, Languages and Programming*, ser. Lecture Notes in Computer Science, W. Brauer, Ed. Springer Berlin Heidelberg, 1985, vol. 194, pp. 465–474.

[191] C. R. Spitzer, *The avionics handbook*. CRC Press LLC, 2001.

[192] J. Sztipanovits, "Composition of cyber-physical systems," in *Proc. IEEE Int. Conf. and Workshops on Engineering of Computer-Based Systems*, March 2007, pp. 3–6.

[193] J. Sztipanovits and G. Karsai, "Model-integrated computing," *IEEE Computer*, pp. 110–112, 1997.

[194] A. Tiwari, "Abstractions for hybrid systems," *Formal Methods in System Design*, vol. 32, no. 1, pp. 57–83, 2008. [Online]. Available: http://dx.doi.org/10.1007/s10703-007-0044-3

[195] C. Tomlin, J. Lygeros, and S. Sastry, "A game theoretic approach to controller design for hybrid systems," *Proc. IEEE*, vol. 88, no. 7, pp. 949–970, July 2000.

[196] J. Torn, "On the hardness of graph isomorphism," *SIAM Journal on Computing*, vol. 33, no. 5, pp. 1093–1108, 2004.

[197] S. Tripakis, B. Lickly, T. A. Henzinger, and E. A. Lee, "A theory of synchronous relational interfaces," *Trans. on Programming Languages and Systems*, vol. 33, no. 4, 2011.

[198] S. Uckun, "META II: Formal co-verification of correctness of large-scale cyber-physical systems during design," Tech. Rep., Sep. 2011. [Online]. Available: http://www.darpa.mil/uploadedFiles/Content/Our_Work/TTO/Programs/AVM/PARC META Final Report.pdf

[199] D. A. van Beek, W. Fokkink, D. Hendriks, A. Hofkamp, J. Markovski, J. M. van de Mortel-Fronczak, and M. A. Reniers, "CIF 3: Model-based engineering of supervisory controllers," in *Proc. Int. Conf. Tools and Algorithms for the Construction and Analysis of Systems*, E. Ábrahám and K. Havelund, Eds. Springer, 2014, pp. 575–580. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-54862-8_48

[200] F. Wang, "RED: Model-checker for timed automata with clock-restriction diagram," in *Workshop on Real-Time Tools, Aug*, 2001, pp. 2001–014.

[201] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra *et al.*, "The worst-case execution-time problemoverview of methods and survey of tools," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 7, no. 3, p. 36, 2008.

[202] J. C. Willems, "The behavioral approach to open and interconnected systems," *Control Systems Magazine*, pp. 46–99, 2007.

[203] W. L. Winston, *Operations Research: Applications and Algorithms, 4th Edition*. Independence, KY: Cengage Learning, 2004.

[204] E. M. Wolff, U. Topcu, and R. M. Murray, "Optimization-based trajectory generation with linear temporal logic specifications," in *IEEE International Conference on Robotics and Automation (ICRA)*, May 2014, pp. 5319–5325.

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

[206] H. Wong-Toi, "The synthesis of controllers for linear hybrid automata," in *Int. Conf. Decision and Control*, vol. 5, Dec. 1997, pp. 4607–4612.

[207] T. Wongpiromsarn, U. Topcu, and R. M. Murray, "Formal synthesis of embedded control software for vehicle management systems," in *AIAA Infotech@Aerospace*, 2011.

[208] T. Wongpiromsarn, U. Topcu, and R. Murray, "Automatic synthesis of robust embedded control software," *AAAI Spring Symposium on Embedded Reasoning: Intelligence in Embedded Systems*, 2010.

[209] T. Wongpiromsarn, U. Topcu, N. Ozay, H. Xu, and R. M. Murray, "TuLiP: a software toolbox for receding horizon temporal logic planning," in *Proc. Int. Conf. Hybrid Systems: Computation and Control*. New York, NY, USA: ACM, 2011, pp. 313–314.

[210] M. D. Wulf, "From timed models to timed implementations," Ph.D. dissertation, Universite Libre de Bruxelles, 2006-7.

[211] H. Xu, U. Topcu, and R. M. Murray, "A case study on reactive protocols for aircraft electric power distribution," in *Int. Conf. Decision and Control*, 2012.

[212] S. Yovine, "KRONOS: a verification tool for real-time systems," *Int. J. Software Tools for Technology Transfer*, vol. 1, no. 1-2, pp. 123–133, 1997. [Online]. Available: http://dx.doi.org/10.1007/s100090050009