Automating Contract-based Design for Cyber-Physical Systems

Sheng-Jung Yu

Electrical Engineering and Computer Sciences
University of California, Berkeley

Technical Report No. UCB/EECS-2025-84
http://www2.eecs.berkeley.edu/Pubs/TechRpts/2025/EECS-2025-84.html

May 16, 2025

Automating Contract-based Design for Cyber-Physical Systems

By

Sheng-Jung Yu

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 Sangiovanni-Vincentelli, Chair
Professor Sanjit Seshia
Associate Professor Anil Aswani

Spring 2025

Automating Contract-based Design for Cyber-Physical Systems

Abstract

Automating Contract-based Design for Cyber-Physical Systems

by

Sheng-Jung Yu

Doctor of Philosophy in Engineering - Electrical Engineering and Computer Sciences

University of California, Berkeley

Professor Alberto Sangiovanni-Vincentelli, Chair

Cyber-physical systems (CPS), which integrate computational and physical processes, present challenges in modeling, specification, and integration due to their heterogeneous nature and complex interactions. Contract-based design aims to address these challenges by using formal specifications to support hierarchical decomposition and system-level reasoning through contract manipulations. Combining this methodology with design automation, which leverages computational power to streamline design tasks, offers a promising approach to addressing the CPS design challenges.

This dissertation focuses on automating the contract-based design process to facilitate its application in cyber-physical system design. We identify the key design automation needs for contract-based design as specification, verification, simulation, and synthesis. Specification enables the expression of requirements and implementations as contracts while assisting in their manipulation. Verification detects potential errors in the decomposition process, ensuring a correct-by-construction design. Simulation provides insight into formal specifications by generating behaviors allowed by their semantics, helping designers confirm that contracts align with design intent and component characteristics. Synthesis automates the decomposition process and optimizes the design.

We address these needs by bridging key gaps in contract-based design automation. For specification, a new contract formalism, constraint-behavior contracts, is introduced to represent physical components using implicit equations, enabling precise expression of requirements. Verification techniques based on receptiveness and strong replaceability, a newly proposed contract relation, are developed to detect decomposition errors, including in feedback systems, ensuring correct-by-construction designs. We also propose a simulation framework that generates behaviors allowed by contract semantics and efficiently produces a small yet insightful set of examples to aid in validating contracts and localizing potential specification errors. A component selection algorithm combining black-box optimization with contract-based system reasoning is proposed to incorporate behaviors into the decomposition process

and enable contract-based synthesis that addresses optimization objectives involving behaviors.

We integrate these contributions into ContractDA, the first tool for contract-based design to offer comprehensive design automation support, including specification, verification, simulation, and synthesis. ContractDA incorporates the proposed functionalities along with existing contract manipulations, offering an interface that enables designers and researchers to effectively apply contract-based design.

To my beloved parents.

Contents

List of Figures

List of Tables

List of Algorithms

Acknowledgments

This dissertation marks the culmination of my journey as a student at Berkeley. The path has never been easy——filled with uncertainty and unpredictability——but also with joys and the support of many people, for which I am deeply grateful. It began during an unprecedented pandemic, a time of shutdowns, minimal human contact, and widespread travel restrictions. Plans often had to change abruptly to adapt to new circumstances. Now, as I write this acknowledgment, the pandemic has become a thing of the past, yet uncertainty and unpredictability persist. Heightened geopolitical tensions and the rapid evolution of technology often leave people wondering whether a revolutionary shift, whether for better or worse, is on the horizon. Fortunately, throughout these five years, I have received immense support from many people, helping me navigate this challenging journey and complete this work. I would like to extend my heartfelt gratitude to all who have supported me along the way.

First, I would like to express my deepest gratitude to my advisor, Professor Alberto Sangiovanni-Vincentelli, for his invaluable support, academic guidance, and holistic vision drawn from his outstanding expertise. Alberto welcomed me into his team, fostering an inclusive and supportive research environment with a balance of professionalism and approachability. He was always available to offer guidance, carefully review my results and manuscripts, and encourage me during setbacks, such as unsuccessful paper submissions. His insightful advice, spanning technical, industrial, and academic perspectives, has been instrumental in shaping both my research and personal growth.

I would also like to thank Prof. Sanjit Seshia and Prof. Anil Aswani for being members of my dissertation committee. The collaboration project with Sanjit during the first half of my PhD journey greatly shaped the foundation of this work. I am grateful for the valuable feedback from the committee members, whose insights and expertise have highlighted areas for improvement and have helped enhance the quality of this dissertation. Additionally, I would like to thank Edward A. Lee for chairing my qualification examination committee.

I would like to express my sincere appreciation to my colleague, Inigo Incer, for his expertise and incisive views in the field. Our fruitful and enjoyable discussions on contracts significantly enriched this work. I also want to thank Shaokai Lin for being my research companion and for shaping our shared interests in research directions. Our collaboration on course projects formed the foundation for the ideas presented in this work. I am also grateful to Baihong Jin and Xianyu Yue for their support during the pandemic and for their helpful advice throughout my PhD studies. Additionally, I want to thank the members of Professor Alberto's research group, Zheng Liang, Tung-Wei Lin, and Matteo Guarrera, for their genuine support and camaraderie throughout my studies.

I would also like to thank the University of California, Berkeley, for providing the fellowship that supported my first two years of study. My sincere thanks go to Professor Pierluigi Nuzzo for offering me the opportunity to serve as a GSI for a renovated class when I needed funding the most. I am also grateful to Judy Ileana Smithson, Shirley Salanio, and Susanne Kauer for their invaluable administrative support. Their prompt responses in clarifying de-

partment policies and assisting with administrative matters made my journey through each milestone a smooth and welcoming experience.

I am fortunate to have met many Taiwanese friends in the Bay Area. I am grateful for their companionship beyond the realm of research, whether through traveling, organizing events, or exploring new hobbies such as snowboarding. To name but a few, listed in alphabetical order: David Chang, Oscar Chen, Pei-Wei Chen, Reichi Chen, Vincent Chen, Jonathan Chou, Gavin Lee, Jennifer Lin, Jia-An Lin, Yen-Cheng Lin, Lily Sheu, Yi-Chi Sheu, Jiyun Tsai, Audrey Wang, Judy Wu, and Issac Yu. You have made my time in the Bay Area an enjoyable and memorable journey. I am also thankful to my undergraduate and high school friends for staying in touch, whether through in-person reunions in the United States or remote conversations. To name a few: David Fan, JiunAn Fan, Jerry Ho, Guo-Liang Hong, Weiyuan Hsieh, Yung-An Hsieh, Chia-Han Huang, Chien-Yu Huang, Jeremy Jahn, Chen-Chien Kao, Sky Kuo, Tony Liang, Thomas Mao, Willy Tai, Eric Wang, Chun-Yen Yao. Your support and encouragement have kept me motivated throughout this journey.

Finally, and most importantly, I would like to express my deepest gratitude to my beloved family for their unwavering support and encouragement across the Pacific Ocean. Their love and strength were especially meaningful during the pandemic, a time of separation and uncertainty. I am profoundly grateful to my father, mother, and sister, who set aside time each week to talk with me about everything, providing comfort and connection despite the distance. I could not have reached this milestone without their belief in me, and this accomplishment is as much theirs as it is mine.

Chapter 1

Introduction

This introductory chapter presents the motivation for this dissertation. We first discuss the design challenges of cyber-physical systems, followed by an introduction to contract-based design methodology as a promising approach to addressing these challenges. The chapter then highlights the role of design automation in facilitating design processes and leveraging computational techniques. Finally, an overview of the key contributions and the organization is provided.

1.1 Cyber-Physical System Challenges

Cyber-Physical Systems (CPS) consist of computational and physical components whose behavior depends on their interaction. Cyber components involve logical operations and communication mechanisms, including software, networks, and algorithms. Physical components encompass tangible elements with shapes, mass, and physical presence that can be observed directly, such as camera lenses, motors, batteries, or pedals. These physical components connect the system with the physical world by sensing information and actuating responses to influence the environment to achieve specific goals. Cyber and physical components are integrated as a CPS that interacts continuously with the environment [99]. Examples of CPS include autonomous vehicles, aircraft systems, power generation and delivery, robotics, and medical devices. The advancement of CPS has led to its ubiquity, significantly enhancing efficiency and comfort in daily life. Consequently, streamlining the CPS design process is essential for driving further advancements, paving the way for an even more efficient and comfortable world [157].

CPS are built on the foundation of components from various separately developed domains, including control algorithms, processor architecture, modern memory technology, sensors and actuators, and complex networks for communication. The heterogeneous nature and complex interactions between these components result in a prolonged and error-prone design process that leads to prohibitively high costs. For example, a commercial jet typically requires five to ten years of development, from design and testing to production, before it is

available on the market, while an automotive model may take three to four years to be rolled out. The overwhelming costs associated with the lengthy development process discourage industry involvement and impede progress in these domains. Furthermore, design faults or supply availability issues can incur additional costs and have a significant impact on the reliability of the system. For example, delays in the development of the Boeing 787 resulted in an estimated $3.3 billion toll, in addition to its wider impact on the industry. Similarly, Toyota's infamous recall of approximately 9 million vehicles due to sticky accelerators highlights the consequences of design faults.

Overall, the CPS design process faces three main challenges caused by heterogeneity and complexity: *Modeling Challenge*, *Specification Challenge*, and *Integration Challenge*. Modeling involves creating a framework that allows for the evaluation and prediction of the systems without relying entirely on experiments conducted after the components are manufactured, thus reducing design costs and time. Specifications define the requirements for the design. They can be expressed informally, such as "The car should decrease its speed when it is foggy," or formally, as in "The adder satisfies $y = a + b$," where y is the output value and a and b represent the input values. Finally, integration involves connecting the manufactured components to work collaboratively and concurrently, resulting in the implementation of the design.

The remainder of this section details the challenges in the cyber-physical system design process, which call for novel methodologies and tools to address them.

1.1.1 Modeling Challenge

As introduced, modeling techniques are essential for reducing design costs and time by minimizing reliance on trial and error with manufactured components. Model-based design [167, 156, 118], which advocates for the systematic use of models, has become a well-accepted practice in the design process. CPS modeling focuses on how such a framework can be established to accommodate heterogeneous components. Two main challenges in CPS modeling are 1. flexibility to balance fidelity with complexity and 2. integrating multiple system concerns.

Balancing model fidelity and complexity is crucial to ensure both the correctness and efficiency of the design process. Model fidelity refers to the degree to which a model represents reality, while model complexity impacts the efficiency of evaluation and prediction. A low-fidelity model that overlooks essential aspects of a component can lead to inaccurate predictions, potentially causing the design to fail to meet the requirements once the components are manufactured. For example, if network and computation delays are ignored in an autonomous vehicle model, the resulting vehicle may struggle to avoid accidents due to slow responses to environmental changes. On the other hand, an overly complex model can reduce the efficiency of evaluation and prediction, leading to a prolonged design process. Additionally, the balance between fidelity and complexity may need to shift throughout the design process, as certain details can be omitted initially and reintroduced later for improved efficiency. This necessitates flexibility in modeling to accommodate varying levels of fidelity

and complexity at different stages of the design process. Thus, a significant challenge is enabling this flexibility and finding optimal balance points for models that meet the needs of each stage of the design process.

Furthermore, because CPS contains both cyber and physical components, models for these components must be able to capture interactions between them. Traditionally, computation is modeled using logic to represent functionality with sequential and discrete semantics. In contrast, physical components often require continuous and concurrent models, such as algebraic differential equations. The differences in these modeling approaches for heterogeneous components present a significant challenge when integrating multiple system concerns in CPS design.

1.1.2 Specification Challenge

Specification, as the starting point for a design, has a profound impact on the design process. The specification challenge involves creating formal specifications and their management.

A vague or ambiguous specification can lead to misunderstandings and misinterpretation of requirements, creating difficulties in analysis and verification and ultimately resulting in a design that fails to align with the original intent. For this reason, formal specifications—based on mathematical techniques with well-defined syntax and semantics—are preferable, as they clearly distinguish acceptable implementations from unacceptable ones. However, formal specifications are often lacking in common design practice, with many requirements still expressed in natural language. For example, the following excerpt is from the Universal Chiplet Interconnect Express (UCIe) specification [159, 170]:

" *Active State transitions: RDI SM must be in Active before Adapter LSM can begin negotiation to transition to Active. Adapter LSM must be in Active before vLSMs can begin negotiations to transition to Active.* "

The requirements use natural language expressions such as "begin negotiation" and "transition to", which are vague in context. These vague expressions increase the burden of interpretation and pose a risk of misinterpretation in the design process. Formulating requirements into formal specifications is a non-trivial task, requiring both finding suitable mathematical techniques that are sufficiently expressive for the design goal and correctly expressing design intent in the chosen techniques. As a result, formulating formal specifications remains a challenge in complex and heterogeneous CPS design.

Furthermore, specifications are shared across various stakeholders, including companies, departments, teams, and designers, throughout the design process, making efficient and accurate requirement management essential to avoid errors. Currently, specifications are divided into chapters, aspects, and viewpoints, with each team focusing on sections relevant to their responsibilities. However, since specification elements are often interdependent, a design choice made by one team may affect others or even cause compatibility issues. The lack of effective requirement management in handling formal specifications thus presents a significant challenge in CPS, particularly in integrating fragmented specifications.

1.1.3 Integration Challenge

The CPS integration challenge lies in ensuring connections compatibility and predicting the outcome of the integration.

Integration brings together all the developed and manufactured components into the final implementation. During this process, all components must function together correctly and produce the desired results. However, issues such as unconnected ports or mismatched ports of the same type may go undetected by design tools, leading to integration errors. Therefore, ensuring compatibility during integration is a critical challenge in CPS design.

Additionally, predicting the outcome before integration occurs presents a significant challenge. Since physically integrating components can be costly and time-consuming, it is crucial to use models that allow us to predict the results of design choices. However, this prediction is difficult due to the heterogeneous nature of the underlying models, as discussed in the modeling challenge. This combined challenge highlights the need to develop methods for evaluating and predicting design impacts through model-based integration before physically integrating the components.

1.2 Design Automation

Design automation, a field of engineering that leverages computational power to accelerate and optimize the design process, is a promising approach to streamline the design process and tackle the CPS design challenges. It focuses on two main objectives: 1. Accelerating design steps, particularly those that would take human designers a long time, and 2. Optimizing design quality by efficiently exploring the design space using theoretical insights and developed algorithms. These objectives are typically achieved by identifying automation needs in the design process, formulating corresponding problems, developing theories and algorithms to solve them, and integrating the solutions into software tools to support design. Design automation is also commonly referred to as computer-aided design (CAD), highlighting the role of computers in automating the design process. Various design fields have already incorporated design automation to streamline their design processes, including mechanical engineering [141, 9], printed circuit board (PCB) design [27], and very-large-scale integration (VLSI) circuit design [172]. Recently, emerging technologies such as microfluidic chips [70], quantum computing [163], and silicon photonics [26] are also driving the demand for new design automation solutions.

Design automation problems are broadly categorized into specification, simulation, verification, and synthesis, which collaboratively enhance design efficiency and quality by working together and influencing each other. Specification defines the solution space and design goals by translating domain knowledge—such as key concerns and material characteristics—into mathematical expressions for computational analysis. Simulation predicts design outcomes, aiding in evaluating design quality and identifying potential faults before manufacturing and integration. Verification ensures correctness by comparing an implementation, whether in-

termediate or final, against the specification to detect errors in the design process. Synthesis explores the design space and applies optimization techniques to generate implementations that meet design goals.

We illustrate these concepts using electronic design automation (EDA) for VLSI circuit design as an example, which is one of the most successful applications of design automation. Electronic design automation emerged in the early 1970s and rapidly developed in the following decades to address the complexity of integrated circuit designs, which involved ever-increasing numbers of transistors and advancing semiconductor technologies [150]. Today, EDA has become an indispensable part of the integrated circuit design process. Design companies purchase licenses from EDA tool companies and develop their in-house automation tools to meet the need for efficient delivery of new products.

In EDA, specifications are captured from targeted algorithms, functionalities, and the required properties that the circuit is intended to perform. These are usually described in C programming language, hardware description languages (HDLs), and SystemVerilog Assertions. Verilog or SystemVerilog at the register-transfer level (RTL) are common HDLs supporting behavioral specifications, considering the data flow between registers, the components used for storing data.

After design specifications are captured, simulation, verification, and synthesis are iteratively performed to generate the design of integrated circuits—the layout masks used to fabricate the chips—from the specification. For example, RTL simulation and verification are performed to ensure that the specified functionality meets the design requirements and satisfies the desired properties. The C programming language is translated into HDLs using high-level synthesis techniques. The RTL description in HDLs is then further synthesized into a gate-level netlist, where the design is represented by the interconnections of logic gates that implement Boolean functions using available standard cells provided by the manufacturing foundry. Gate-level simulation and verification are employed to ensure that the synthesized netlists are error-free and likely to meet design goals, including functionality, timing, power, area, and cost.

Afterwards, physical design synthesizes the gate-level netlist into a mask layout while considering all physical aspects of the circuit. The process consists of multiple stages: partitioning, floorplanning, power planning, placement, clock-tree synthesis, and routing, each progressively adding details to the layout to complete the design. For example, placement determines the locations of standard cells in the layout, while routing establishes the interconnections between the cells. Signoff tools perform comprehensive simulations and optimizations, ensuring that timing and power requirements are met. Finally, physical verification, including design rule checking (DRC) and layout versus schematic (LVS) checks, ensures that the final layout matches the gate-level netlist and is free of serious manufacturing issues. Each step in EDA relies on intricate theories, algorithms, and software tools to facilitate VLSI circuit design, demonstrating the potential of engineering in streamlining the design process.

EDA has been highly successful in enabling efficient and scalable design processes for VLSI circuit design. However, in the domain of CPS, a unified design methodology is still

lacking, with each domain often handling design independently, making holistic automation support for the entire design cycle challenging [149, 158]. Inspired by the success of EDA, this dissertation takes a holistic approach to CPS design automation, using *contract-based design* as the framework to structure and formlize the design process. The next section introduces contract-based design and its role in addressing the CPS design challenges.

1.3 CPS Design Methedology

Design methodology is a systematic approach developed based on prior experience, theory, and available tools to address design challenges. It aims to overcome recurring issues through a structured series of steps that guide the design process. In response to CPS design challenges, various CPS design methodologies have been proposed or adapted from other domains, including the waterfall model, incremental model, V-model, spiral model, and agile model. Each methodology offers distinct advantages and drawbacks. For instance, the waterfall model guides the design process into a sequential, dependent workflow. While it is simple and easy to manage, it struggles to accommodate changing requirements and often delays the delivery of a working solution. In contrast, the spiral and agile models, originating from software engineering, emphasize shorter cycles for early product evaluation. Although effective for iterative feedback, they can lead to higher costs in CPS design due to the expense and time required for physical prototyping compared to software development.

Some industries, such as automotive, have adopted the V-model, as shown in Figure 1.1. The methodology consists of a top-down approach that decomposes the problem into manageable parts, followed by a bottom-up integration, verification, and validation phase [17]. The process begins with defining the design concept and system requirements, including functional aspects that specify the system's intended behaviors and non-functional properties such as power, cost, and delay constraints. These requirements are translated into a high-level architecture reflecting the system's requirements. The architecture is further decomposed into subsystems manageable for development, such as electrical, mechanical, microelectronic, software, and network components. Once individual components are developed and tested, the integration phase begins, where components are assembled into a complete system. Finally, the system undergoes verification and validation to ensure that it meets the original design concept.

However, this oversimplified methodology fails to effectively address the CPS design challenges. First, the waterfall-like approach struggles with scalability. As the design scale increases, this method requires a lengthy turnaround time since issues can only be identified and addressed after integration, resulting in wasted development time and increased costs. Second, design space is often inadequately explored. The transition from requirements to architecture and subsystems is typically guided by heuristics and prior experience, which can lead to suboptimal designs due to the limited exploration of the design space. Given the increasing scale and complexity of CPS, novel design methodologies are essential to meet the growing demands of future technologies.

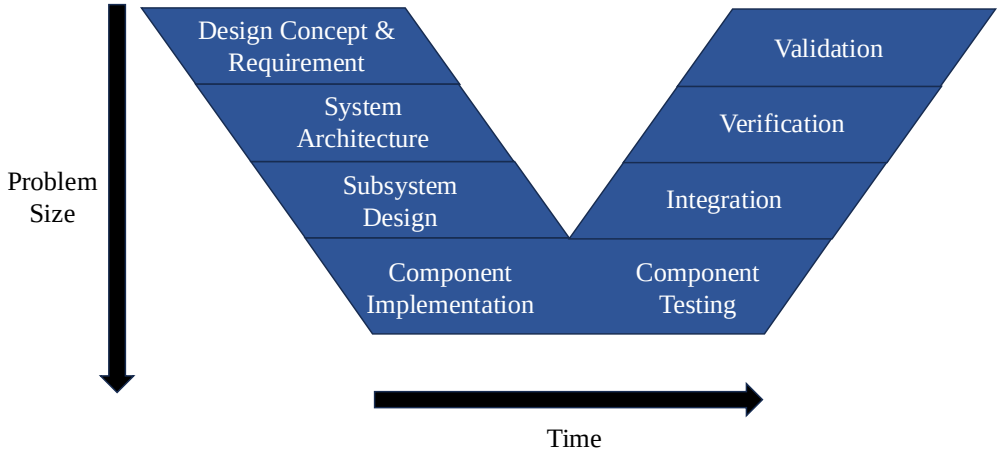


Figure 1.1: The V-model design methodology.

Platform-Based Design (PBD) [87, 149] was introduced to address the limitations of the V-model design process. A platform defines a set of architectures that can be constructed using a library of components, following specific composition rules. Platforms establish abstraction layers, where each layer hides unnecessary details while summarizing key implementation information. The PBD design process consists of both bottom-up and top-down approaches, forming a "meet-in-the-middle" methodology that balances design flexibility and implementation constraints. In the bottom-up process, a platform is built by creating a library of components and modeling their associated performance abstractions. In the top-down process, requirements are mapped onto the components available on the platform. As a result, the two processes meet in the middle, where the requirements is mapped to the platform and characterizations of potential implementations. Once a design is finalized within a platform, the process moves to a lower platform level, which introduces more implementations details. By structuring the design process in this way, the platform reduces complexity, facilitating integration and verification of the implementation. For example, in digital circuit design, logic gates form a platform A Boolean function can be mapped to a set of logic gates with logical connections, without considering lower-level circuit details such as transistors or wiring. Once this mapping is completed, the process moves to the next platform level, which typically consists of standard cells provided by the foundry, incorporating transistor-level details into the implementation.

1.3.1 A Preview on Contract-based Design Methodology

Contract-based design emerges as a promising approach to addressing CPS design challenges. This methodology advocates using contracts to enable formal methods and compositional design throughout the design process. Contracts are a class of formal specifications typically defined by the environments in which the system operates and the properties it must satisfy under those environments. They provide a means to express system requirements and establish well-defined constraints to guide the design process.

The contract-based design methodology adopts compositional design, a divide-and-conquer approach that repeatedly decomposes system contracts into subsystem contracts, addressing the limitations of the V-model while preserving its advantages. As illustrated in Figure 1.2, consider a scenario where contracts are defined for the top-level system ($\mathcal{C}_s$) and its decomposition into several subsystems ($\mathcal{C}_1$, $\mathcal{C}_2$, and $\mathcal{C}_3$). These contracts ensure that any valid implementation of the subsystems can be integrated as a valid implementation of the top-level system. This approach reduces a complex and heterogeneous design problem into manageable, single-domain design problems, similar to the advantage of the V-model. The design process can then proceed hierarchically, from the top-level system down to individual components. Moreover, the use of formal specifications enables formal methods to analyze the relationships between contracts, allowing early integration tests and verification to detect design mistakes, thus addressing the V-model's scalability issue. Design correctness can be ensured through two key aspects:

1. **Correctness in decomposition**: Ensuring that decomposed contracts satisfy higher-level contracts, assuming subsequent decomposition and development are error-free.

2. **Correctness of development**: Verifying that implementation results conform to their corresponding contracts.

By verifying contracts before component development, decomposition errors can be identified early, preventing unnecessary costs and delays. Since this verification process combines contracts as if developers were integrating actual components, it is often referred to as an early integration test, as it provides insight into integration results before component development begins. Additionally, formal specifications rely on precise mathematical formulations, which define well-structured design spaces and evaluation metrics. This enables comprehensive design space exploration beyond heuristics or prior experience, ensuring that optimal solutions are considered.

Contract-based design can serve as a rigorous approach to applying the platform-based design concept. In the bottom-up process, components in a platform are characterized by their contracts. In the top-down process, high-level system contracts are decomposed into subsystem contracts that align with the available component contracts in the platform. Each subsystem contract corresponds to a component in the library, completing the mapping of requirements to the available components. As a result, contract-based design retains the benefits of platform-based design while addressing the limitations of the V-model.

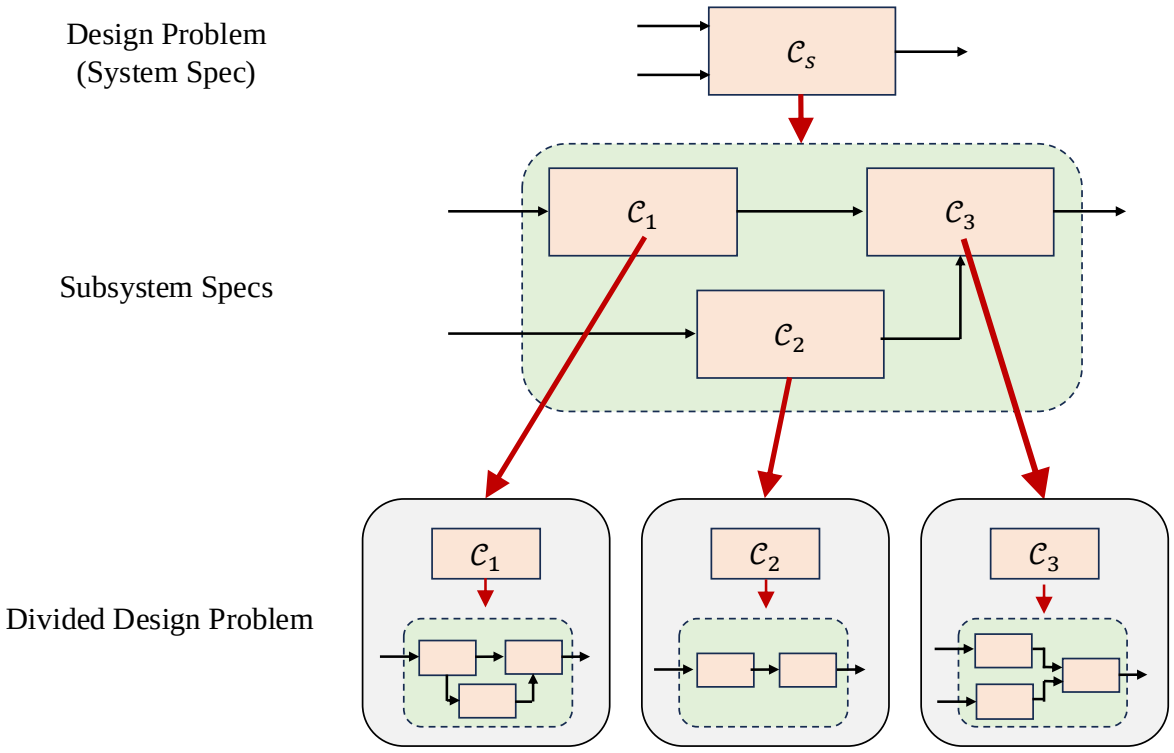


Figure 1.2: The core concept in contract-based design methodology.

The methodology shows promise in addressing the aforementioned CPS modeling, specification, and integration challenges. First, formal specifications require systems and components to be modeled using formal languages, providing a structured approach to balancing fidelity with complexity while integrating multiple system concerns. The notions of refinement and abstraction [7, 10, 11] provide flexibility in balancing fidelity and complexity. Refinement and abstraction are relative concepts: if a specification $\mathcal{C}_r$ is a refinement of another specification $\mathcal{C}_a$, then specification $\mathcal{C}_a$ is an abstraction of $\mathcal{C}_r$. Conceptually, $\mathcal{C}_r$ is a refinement of $\mathcal{C}_a$ if $\mathcal{C}_r$ can safely replace $\mathcal{C}_a$ in the design process without without violating any of the requirements from $\mathcal{C}_a$. Typically, $\mathcal{C}_r$ imposes additional restrictions on top of $\mathcal{C}_a$, such as operating in a broader set of environments or exhibiting a more constrained set of behaviors under those environments. Refinement and abstraction thus involve adding or removing information in the formal specification, resulting in different levels of abstraction. A more abstract contract reduces complexity but sacrifices fidelity, while a more refined contract increases precision at the cost of greater complexity. This ability to shift across abstraction levels provides a structured approach to balancing fidelity and complexity. Furthermore, various operations have been proposed to reason about interactions between systems and various design concerns based on their specifications. For example, composition and merg-

ing are two crucial operators in contract-based design. The composition of specifications produces a new specification that is guaranteed to be satisfied by any system created by integrating subsystems developed under the composed specification. Merging specifications creates a new specification for a system, where each of the merged specifications represents a different viewpoint, such as timing, power, and functionality. These operations enable the precise capture of interactions between and within components by leveraging established reasoning methods.

Secondly, the use of formal specifications inherently calls for solutions to formulate precise specifications and facilitates the management of requirements. Once all components, requirements, and intermediate subsystems are expressed as formal specifications, the risk of vague and ambiguous design requirements is mitigated, enabling rigorous verification and validation. The use of contracts throughout the design process also supports specification management. By expressing relationships between requirements such as refinement, abstraction, composition, and merging, the connections between different requirements can be easily reviewed and verified. This approach makes the process less error-prone and more transparent for all stakeholders.

Furthermore, integration challenges can be addressed through early integration tests. Compatibility issues during integration can be mitigated by detecting them during the decomposition process, and the use of formal specifications streamlines the examination of compatibility. The outcomes of integration can also be predicted before physical integration occurs, as these specifications can be virtually integrated and evaluated through operations, without waiting for the development or manufacturing of components. This enables faster validation and evaluation of the design, reducing both design costs and time spent on suboptimal or incorrect designs that fail to meet the requirements.

1.4 Dissertation Overview

Given the potential of contract-based design and design automation, this dissertation focuses on design automation techniques, including theories, automation methodologies, algorithms, and tool development, to streamline contract-based design methodology for CPS.

First, the key design automation tasks for contract-based design are identified as specification, verification, simulation, and synthesis. The dissertation then proposes techniques to bridge existing gaps and achieve these tasks:

- **Specification**: A new contract formalism for physical components is introduced, enabling the use of implicit functions and their integration with existing formalisms, making contract-based design more applicable to CPS.
- **Verification**: Theories and algorithms are developed to ensure that decomposed contracts enable independent development without introducing integration issues.

- **Simulation**: A novel methodology and algorithms are proposed to help ensure that formulated contracts align with design intent and component characteristics.
- **Synthesis**: A contract synthesis algorithm is devised to decompose contracts into a set of library contracts when system behavior must be considered in design objectives.

Finally, an automation tool is developed to integrate these tasks, providing a comprehensive design automation platform for contract-based CPS design.

1.5 Main Contributions

This section summarizes the dissertation's main contributions, categorized in theory, methodology, algorithms, and tool development.

1.5.1 Theory

Contract theory forms the foundation of contract-based design methodology and is essential for developing algorithms for design automation. This dissertation bridges two major theoretical gaps to ensure the methodology can be applied to cyber-physical systems without introducing design faults: formalism for physical systems and conditions for correct contract decomposition.

1.5.1.1 Contract Formalism for Physical Systems

This dissertation proposes *constraint-behavior contracts*, a new contract formalism that expresses physical components through implicit functions, preserving their physical meaning without specifying port directions. Implicit functions and the absence of port directions are common in the modeling of physical systems [176]. However, existing contract frameworks, such as assume-guarantee contracts and interface I/O automata, require contracts to be expressed with explicit functions reflecting designated port directions, making it difficult to represent physical component specifications. Specifically, Chapter 4 introduces the formalism and derives its properties and operations. The formalism helps designers avoid faults in formulating contracts, as it aligns with their intuitive understanding of components. Additionally, constraint-behavior contracts can be easily converted into assume-guarantee contracts without loss of information. Ultimately, this discovery leads to a methodology for combining physical and cyber components using different contract formalisms.

1.5.1.2 Conditions for Correct Contract Decomposition

This dissertation also addresses a theoretical gap in defining correct contract decomposition. Previous work on contracts suggests using refinement as the criterion for contract decomposition. However, this dissertation demonstrates that a set-based definition of contracts

allows vacuous systems—systems lacking any behavior—to be developed under contracts using refinement as the decomposition criterion. This issue arises particularly when contracts are refined independently without immediate integration testing, leading to difficulties in leveraging the benefits of compositional design to reduce complexity. Chapter 5 formalizes this issue as the *vacuous implementation* problem and introduces constraints on top of refinement to ensure correct contract decomposition. The constraints include the concept of *Strong Replaceability* and graph-based properties derived from the relationships between the decomposed contracts. Strong replaceability requires that refined contracts preserve receptiveness with respect to all target environments, while the graph properties establish conditions to ensure strong replaceability for any independent refinement in feedback composition. These theories resolve the issue in assume-guarantee contract decomposition, ensuring the benefits of compositional design in contract-based design.

1.5.2 Methodology

In addition to theoretical contributions, this dissertation advances the contract-based design methodology by promoting the integration of physical components into the design process, proposing simulation methodology to verify if contracts align with design intent, and identifying the requirements for contract-based design automation tools.

1.5.2.1 Methodology for Combining Physical Components and Cyber Components

Building on the theory of constraint-behavior contracts, Chapter 4 develops a methodology for integrating physical devices with cyber components. In this approach, designers can select contract formalisms for specifications based on factors such as reusability, ease of expression, and the nature of the components. As abstraction levels change, designers can convert the specifications into the formalism best suited to the needs at each level leveraging the ease of conversion between assume-guarantee contracts and constraint-behavior contracts. This methodology is demonstrated through a UAV propulsion system design problem, showcasing its effectiveness in verifying that the design satisfies the system specifications.

1.5.2.2 Simulation for Verifying Contracts

While formal methods can provide correctness guarantees for contract-based design, the specification formulation—created by human designers to reflect their design intent—poses risks to the correctness of the design. Any mistakes in the formulation cannot be detected without comparing its semantics to the design intent, as the specification serves as the gold standard throughout the rest of the design process. To ensure that the design intent is accurately translated into contracts, Chapter 6 proposes using simulation to verify whether the formulated contract aligns with the design intent. By examining the behaviors gener-

ated through simulation, designers can confirm that the contract accurately represents the intended design objectives.

1.5.2.3 Methodology for Contract-based Design Tools

Chapter 3 identifies the need for design automation tools in contract-based design to facilitate the application of design automation techniques in the design process. Through comparisons and analysis of existing tools, the requirements for such automation tools are summarized as support for comprehensive contract manipulations, various design automation tasks, and extensibility to accommodate new formalisms, modeling techniques, and solvers.

1.5.3 Algorithm

Algorithms are the cornerstone of design automation, enabling design tasks to be completed efficiently and effectively. This dissertation advances contract-based design automation by contributing new algorithms including component selection using behaviors, verification of correct decomposition, and simulation for evaluation and debugging.

1.5.3.1 Component Selection using Behaviors

This dissertation introduces the first contract-based component selection algorithm capable of handling constraints and optimizing objectives that involve system behaviors. Component selection is a critical contract synthesis problem, where contracts are decomposed into a set of library components that satisfy specifications while optimizing design objectives.

Existing contract selection algorithms address a simplified problem where objectives can be evaluated independently for each selected component. For example, if cost is the objective, these approaches assume that the total cost can be obtained by summing the individual costs of subsystems. However, design problems often involve subsystem interactions, such as system behaviors that cannot be determined independently. Chapter 7 introduces an algorithm that integrates contract-based system reasoning with black-box optimization to select components while accounting for their interactions to optimize the objective function. This approach expands the capability of contract synthesis to handle more complex design objectives.

1.5.3.2 Verification of Correct Decomposition

As discussed in Section 1.5.1.2, vacuous implementation issues may arise if refinement is used as the sole criterion for verifying correct decomposition. To prevent these issues, automated verification algorithms are essential for checking whether the conditions after the theories for correct contract decomposition are developed, ensuring valid decomposition. Chapter 5 introduces graph-inspired algorithms that verify decomposition correctness, preventing contract decomposition that can lead to vacuous implementation from being used for independent development. These algorithms are abstractly designed to accommodate

any set-based specification with a compatible theorem solver, while their implementability is demonstrated for finite sets and first-order logic contracts.

1.5.3.3 Simulation for Contract-based Design

As discussed in Section 1.5.2.2, simulation is proposed to verify whether formulated contracts align with design intent. Chapter 6 introduces an algorithm for contract simulation, capable of generating behaviors specified by the contracts under environmental constraints. Additionally, the algorithm can automatically produce constraints to help designers examine the correctness of operators used in the formal specification with a small set of simulated behaviors.

1.5.4 Tools

In Chapter 8, we present ContractDA, an automation tool developed for contract-based design. The tool supports comprehensive contract manipulations and design automation tasks while providing abstractions to accommodate new formalisms, modeling techniques, and solvers.

1.6 Organization

This chapter has outlined the motivation for automating contract-based design to address CPS design challenges.

The remainder of this dissertation is organized as follows: Chapter 2 presents the background materials, including system modeling, specification formalisms, contracts, contract-based design, assume-guarantee contracts, and the development history of contract-based design. System modeling and specification formalisms provide the languages for expressing requirements and describing implementation capabilities. Contracts are introduced as an abstract class of formal specifications, followed by a discussion on the advantages of contract-based design. Assume-guarantee contracts, one of the most commonly used contract formalisms, are then presented along with their properties, operations, and relations as a concrete instantiation of contracts. Finally, the development history of contract-based design is reviewed to provide context for its evolution.

Chapter 3 discusses design automation opportunities for contract-based design and reviews existing work. Crucial automation tasks for contract-based design are identified, along with their corresponding design automation problem categories. For each task, the problem formulation and existing work are examined to further highlight research opportunities. Additionally, existing contract-based design automation tools are reviewed, leading to the argument that a new tool is necessary to provide a comprehensive solution for these automation tasks.

Chapters 4 through 7 address gaps in theory, methodology, and algorithms for the identified automation tasks. Chapter 4 introduces constraint-behavior contracts, a new contract formalism for physical components, accommodating models using implicit functions. Theoretical foundations, including properties, operations, and relations, are developed to demonstrate that the formalism is a concrete instantiation of contracts and to highlight its advantages. Additionally, a methodology for integrating different formalisms and converting them based on needs in abstraction levels is presented, demonstrated through a UAV propulsion system design verification problem. This work is based on joint research with Inigo Incer and Alberto Sangiovanni-Vincentelli [181].

Chapter 5 defines correct contract decomposition and verification techniques to ensure that a given decomposition does not introduce issues after independent development. The problem of vacuous implementation is first examined, highlighting the need for correct contract decomposition and the insufficiency of refinement in preventing the issue. Conditions, including receptiveness and graph properties derived from the relationships between the decomposed contracts, are then identified to satisfy these requirements. Verification algorithms are developed to check these conditions and detect incorrect contract decompositions. This chapter is based on joint work with Inigo Incer and Alberto Sangiovanni-Vincentelli [182].

Chapter 6 proposes algorithms for simulation and a methodology for applying simulation to check if the formulated contracts align with the design intent. The methodology ensures that contract-based design does not use incorrect specifications as design goals. An algorithm is introduced to produce behaviors allowed by the contracts under environmental constraints, assisting designers in examining the formulated contracts. Additionally, an automated constraint generation algorithm is proposed to help designers verify if the design intent is met by a small set of behaviors, without the need to specify the environmental constraints. This chapter is based on joint work with Alberto Sangiovanni-Vincentelli.

Chapter 7 proposes a synthesis algorithm for component selection using system behaviors. The need for incorporating system behaviors into the component selection problem is first identified to account for interactions between systems. A contract-based system reasoning framework is designed to produce initial solutions, verify synthesis results, and evaluate design objectives. An algorithm based on a black-box optimization flow using Bayesian optimization is introduced to optimize component selection, demonstrated through a UAV propulsion system design problem. This chapter is based on joint work with Alberto Sangiovanni-Vincentelli.

Chapter 8 presents ContractDA, a contract-based design automation tool that integrates the developed automation tasks. The functionality provided by the tool, including comprehensive contract manipulations and design automation tasks, is introduced. The design of the tool, including its interface and architecture for abstracting contract-based design, as well as its extensibility to accommodate different models, formalisms, and solvers, is presented to highlight the contributions of the tool.

Finally, Chapter 9 concludes the dissertation and discusses future directions for design automation in contract-based design for CPS.

Chapter 2

Preliminaries

This chapter provides the background concepts for this dissertation. First, the formalisms for modeling system behaviors and specifications are introduced to establish the languages for expressing requirements and describing implementation capabilities. Next, the definition of contracts is presented, followed by a discussion on the advantages of using them in contract-based design. Then, assume-guarantee contracts, one of the most commonly used contract formalisms, are introduced along with their properties, operations, and relations as a concrete instantiation of contracts. Finally, the historical development of contracts is reviewed to provide context for their evolution.

2.1 Formalisms for System Modeling and Specification

Modeling provides a mathematical framework for explaining observed physical quantities and describing behaviors, which result from interactions between a component and its environment. It enables predictions for complex systems and establishes a rigorous basis for analysis and guidance in the design process. For example, system models allow simulation under various operating conditions to evaluate system performance. Models can also guide the design process by first specifying desired behaviors and then mapping them to implementations, as demonstrated in the top-down process of platform-based design methodology. For instance, combinational logic and finite-state machines are commonly used to model the behaviors of digital circuits. In the digital circuit design process, the desired system behavior is first described using these models and then mapped to logical gates composed of transistors that implement the specified behavior.

Due to the complexity of physical systems, models are often abstracted to reduce complexity, making computation more efficient while maintaining sufficient accuracy to predict behaviors. For example, Boolean values can represent high and low voltage levels, even though actual voltages may not precisely match the ideal levels, especially during transitions. When the non-ideal values and the transitions before reaching a stable voltage level are ignored, the system can be modeled using Boolean values, enabling the use of Boolean

algebra to facilitate computation, prediction, and design optimization.

Various modeling techniques have been proposed to support system analysis and design. Examples include finite-state machines, actor models [66], communicating sequential processes (CSP) [68], and Kahn process networks [84]. The tagged signal model [98] provides a unified framework that integrates these models, offering a systematic approach to describe system behaviors. This section briefly introduces the tagged signal model and defines the modeling framework used in this dissertation, with simplifications made for clarity in the examples throughout.

In contrast to models, which describe what a system *will* do, formal specifications focus on what the system *should* do as a requirement. While differing in purpose, both center on system behavior, and thus specifications are defined using the framework created by models. Specifically, specifications define the acceptable behaviors of interfaces between systems and their environment as requirements. An implementation of a specification is a model that produces only the acceptable behaviors, thus satisfying its requirements. Internal system behaviors are ignored, as they are implementation details that do not affect the interaction between systems and their environments. Consequently, formal specifications require the modeling of system interface behaviors. The requirements are described through these models, enabling the creation and verification of implementations based on them.

The following introduces formal definitions for both modeling and formal specification as considered in this dissertation.

2.1.1 Ports

Ports serve as the interfaces through which systems interact with their environment, carrying physical quantities or other conceptual values used for modeling and describing system requirements.

Definition 2.1. *A port p is a variable associated with a port type $\mathcal{V}_p$, which represents the set of possible values for the port.*

A port acts as a variable, allowing values to be assigned that represent physical or conceptual quantities. The port type defines the range and type of these values.

Since a system may contain many ports, we define the system ports as the collection of all its ports.

Definition 2.2. *The system ports are denoted as a set $\mathcal{P}$.*

The following examples illustrate the concept of ports.

Example 2.1. *The logical AND gate in Figure 2.1(a) contains three ports: a, b, and c. The port types associated with these ports are the sets of Boolean values, denoted as $\mathbb{B}$. Therefore, $\mathcal{V}_a = \mathcal{V}_b = \mathcal{V}_c = \mathbb{B}$. The system ports of the logical AND gate are denoted by $\mathcal{P}_{ANDgate} = \{a, b, c\}$.*

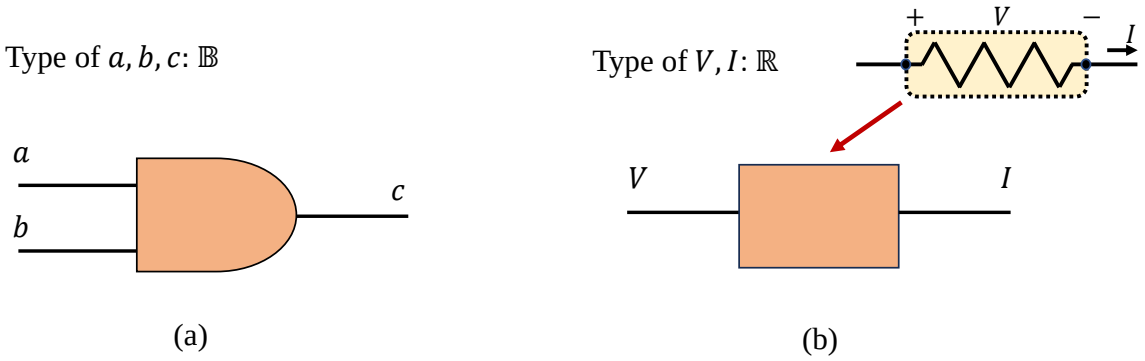


Figure 2.1: Examples of systems to be modeled: (a) a logical AND gate and (b) a resistor.

Example 2.2. *The resistor in Figure 2.1(b) contains two ports: V and I. The port types associated with these two ports are the sets of real values, denoted by $\mathbb{R}$. Therefore, $\mathcal{V}_V = \mathcal{V}_I = \mathbb{R}$. The system ports of the resistor are denoted by $\mathcal{P}_{resistor} = \{V, I\}$.*

Note that a port is not necessarily a physical connector through which a system can be connected to other systems. In the case of the resistor, although its physical shape contains two connectors, the physical quantities of voltage and current are defined through the relationship between the two connectors. Voltage is the electric potential difference between the two connectors, while current is the rate of electrons passing through the system.

2.1.2 Behaviors

As introduced, behaviors are the focus of modeling. The behaviors defined on the system ports enable the understanding of the relationships between port values.

Modeling formalisms can influence the concrete definition of behaviors. In the tagged signal model, a behavior is referred to as a process—a set of events (t, v) containing a tag t and a value v. Thus, a behavior (process) for a port p with its port type $\mathcal{V}_p$ is a set $\{(t, v) \mid t \in \mathbb{T}, v \in \mathcal{V}_p\}$, where $\mathbb{T}$ is the set of available tags. Tags represent the notion of time when $\mathbb{T}$ is a totally ordered set. For instance, when the tag set is $\mathbb{R}$, the system is modeled in continuous time, and when the tag set is $\mathbb{N}$ (the set of natural numbers), it is modeled as a discrete-event system. In such cases, tags are referred to as *time stamps*. Alternatively, tags may be defined over a partially ordered set, in which case they do not correspond directly to a notion of time. Interested readers can refer to Lee and Sangiovanni-Vincentelli [98] for more details on tags and their semantics in modeling. The tagged signal model is a general framework that unifies various models of computation. To facilitate understanding, these preliminaries focus on the case $\mathbb{T} = \mathbb{N}$, a subset of the general model.

Example 2.3. *A process $\{(t = 0, v = True), (t = 3, v = False), (t = 6, v = False)\}$ represents a behavior of the port a in Figure 2.1(a), illustrating discrete changes in value.*

Example 2.4. *A process* $\{(t, v) \mid v = 2t\}$ *represents a behavior of port* V *in Figure 2.1(b), illustrating continuous updates of the voltage values according to the function* $f(t) = 2t$.

When behaviors are modeled as discrete events, with value changes occurring only at specific times, they can be represented simply as a sequence of values, since the timing can be inferred from context. In this dissertation, behaviors represented without time stamps are modeled as discrete events, with the order of values reflecting the sequence of system events.

Example 2.5. *For the logical AND gate behavior in Example 2.3, if discrete events occur at time stamps* 0, 3, *and* 6, *the behavior is represented as* (*True*, *False*, *False*).

These examples also illustrate that modeling formalisms determine the concrete form of behavior definitions.

Furthermore, a behavior can be described either statically or dynamically, depending on the system's characteristics. Dynamic behaviors, as illustrated in the previous examples, capture information about value changes, ordered either by sequence or by time stamps. In contrast, when a system's port relations are independent of previous values, value changes need not be considered, as the current value alone is sufficient to characterize the behavior. In such cases, the time stamps can be omitted, and the behavior is represented by a single value instead of a sequence.

Example 2.6. *In Figure 2.1, since the systems' behaviors do not depend on previous values, they can be represented statically. For the logical AND gate's port* a, *the behavior can be either* $(a = False)$ *or* $(a = True)$. *For the resistor's port* V, *the behavior can be any element of the set* $\{V \mid V \in \mathbb{R}\}$.

Regardless of the modeling formalism, the behaviors of a single port can be represented as elements of a set.

Definition 2.3. *Let* $\mathcal{B}_p$ *denote the universe of behaviors that a port can exhibit, as determined by its port type. A behavior is an element of* $\mathcal{B}_p$.

After introducing port behaviors and the variety of modeling formalisms at the port level, we now extend the discussion to system behaviors. The behavior of a system is the combination of the behaviors of its ports, which collectively define the universe of possible system behaviors.

Definition 2.4. *The universe of system behaviors is defined as the Cartesian product of the universes of the port behaviors, i.e.,* $\mathcal{B}_\mathcal{P} = \prod_{p \in \mathcal{P}} \mathcal{B}_p$, *and a system behavior is an element of* $\mathcal{B}_\mathcal{P}$.

Example 2.7. *Using the notion of static behaviors, the universe of behaviors for the logical AND gate in Figure 2.1(a) is* $\mathbb{B}^3$, *where each tuple corresponds to the values of ports* a, b, *and* c. *Examples of system behaviors include* $(a = false, b = true, c = false)$ *and* $(a = true, b = true, c = true)$.

Example 2.8. *Similarly, using the notion of static behaviors, the universe of behaviors for the resistor in Figure 2.1(b) is* $\mathbb{R}^2$*, where each tuple corresponds to the values of ports* V *and* I*. Examples of system behaviors include* $(V = 4, I = 1)$ *and* $(V = 8, I = 2)$*.*

2.1.3 Properties

Behaviors can be used to define a requirement, called a property, as a collection of behaviors of interest. If a system produces only behaviors contained in the property, it satisfies the property; otherwise, the system is said to violate the property.

Definition 2.5. *A property* P *is a subset of the system behavior universe, i.e.,* $P \subseteq \mathcal{B}_{\mathcal{P}}$*.*

In the formal specification community, properties are often expressed compactly using logic, simplifying their description and enabling efficient computation. We now introduce property representations using first-order logic, linear temporal logic, and relational interfaces. For a more comprehensive introduction, including formal definitions and examples of first-order and temporal logic, interested readers can refer to Chapter 1 of Rosen [147] and Chapter 13.2 of Lee and Seshia [99].

2.1.3.1 First-order Logic

First-order logic uses quantified variables over non-logical objects, such as predicates and functions. A predicate is an expression that evaluates to either true or false, depending on the values of the variables. A function is an expression that maps variable values to another value. For example, the predicate $Q(a, b)$ can be defined to check whether the values of a and b are equal, while the function $f(I)$ can be defined as $f(I) = 4 * I$.

The syntax of first-order logic is defined by terms and formulas. A term is either a variable or a function applied to other terms. A formula, on the other hand, is recursively defined as one of the following:

1. A predicate, such as $Q(a, b)$
2. Equality between terms, such as $V = I$. Note that equality is not typically a built-in relation in first-order logic, but it can be represented using a predicate like $Q(a, b)$.
3. Negation of a formula: $\neg\phi$.
4. Logical binary operations on formulas, such as $\phi_1 \wedge \phi_2$ and $\phi_1 \vee \phi_2$.
5. Quantifiers applied to a formula: if ϕ is a formula and x is a variable, then $\forall x\ \phi(x)$ and $\exists x\ \phi(x)$ are also formulas.

The semantics of first-order logic denote the set of all variable assignments that make a formula evaluate to true.

Example 2.9. *First-order logic can be used to describe a property. Consider the property* $P = c \Leftrightarrow (a \wedge b)$*, where* $a, b, c \in \mathbb{B}$*. The property consists of all triples* (a, b, c) *such that* $c \Leftrightarrow (a \wedge b)$ *evaluates to true. Formally, it can be expressed as the set* $\{(a, b, c) \in \mathbb{B}^3 \mid c \Leftrightarrow (a \wedge b)\}$*. Specifically, the set includes the tuples* $\{(a = false, b = false, c = false), (a = false, b = true, c = false), (a = true, b = false, c = false), (a = true, b = true, c = true)\}$*.*

Another example of using first-order logic to describe system behavior is the *synchronous relational interface* [169]. A stateless synchronous relational interface is defined as a triple (X, Y, ϕ), where X is the set of input variables, Y is the set of output variables, with X and Y disjoint, and ϕ is a first-order logic formula over the variables in $X \cup Y$, specifying the valid behaviors of the interface.

Example 2.10. *Consider a synchronous relational interface where the input set* X *is* $\{I\}$*, the output set* Y *is* $\{V\}$*, and the first-order logic formula* ϕ *is defined as* $(V = 4I)$*. This defines a property where the output voltage* V *must be four times the value of the input current* I*.*

Therefore, in the remainder of the dissertation, when the context surrounding the variables is clear, the first-order logic formula ϕ will be used to denote the set of port values that evaluate ϕ to true, without explicitly writing the corresponding set comprehension. Additionally, *True* will represent the universe of behaviors, $\mathcal{B}_{\mathcal{P}}$, and *False* will represent the empty set, $\emptyset$.

Since first-order logic can describe sets over port variables, it is well-suited for expressing static behaviors, where only port values are considered and time or value changes are irrelevant.

2.1.3.2 Linear Temporal Logics (LTL)

However, when systems require temporal information, first-order logic may become less compact, as it necessitates introducing separate variables for each time stamp.

Linear Temporal Logic (LTL)[137] enables formal reasoning about temporal behaviors in discrete-event systems, such as those described in Example 2.5. These behaviors are expressed as sequences of port values, known as *traces*. A trace is a sequence

$$(q_0, q_1, q_2, \ldots)$$

, where each q_i contains the values for all ports.

LTL is defined on top of propositional logic. Unlike first-order logic, propositional logic does not include functions, predicates, values other than Boolean values, or quantifiers. It can be seen as a special case of first-order logic, with first-order logic building upon the foundation established by propositional logic. Extensions, such as first-order temporal logic, have been proposed, but they tend to suffer from high computational complexity.

In LTL, each propositional logic variable is an LTL formula. The Boolean operations between LTL formulas also form an LTL formula. A trace satisfies an LTL proposition p if

p is true for q_0. In addition, LTL utilizes several temporal operators, which operate on any LTL formula ϕ and result in another LTL formula:

- $\mathbf{G}\phi$: Read as "globally ϕ". A trace satisfies $\mathbf{G}\phi$ if every suffix of the trace satisfies ϕ. A suffix of a trace is the sequence starting from some element q_j in q and including all subsequent elements. For example, the traces $(q_1, q_2, q_3, \ldots)$ and $(q_2, q_3, \ldots)$ are suffixes of the trace $(q_0, q_1, q_2, q_3, \ldots)$. Formally, a trace satisfies $\mathbf{G}\phi$ if and only if for all $j \geq 0$, the suffix $(q_j, q_{j+1}, \ldots)$ satisfies ϕ.

- $\mathbf{F}\phi$: Read as "eventually ϕ". A trace satisfies $\mathbf{F}\phi$ if some suffix of the trace satisfies ϕ. Formally, a trace satisfies $\mathbf{F}\phi$ if and only if for some $j \geq 0$, the suffix $(q_j, q_{j+1}, \ldots)$ satisfies ϕ.

- $\mathbf{X}\phi$: Read as "next state ϕ". A trace satisfies $\mathbf{X}\phi$ if and only if the next state, q_1, satisfies ϕ, where q_1 is the state that immediately follows the initial state q_0 in the trace $(q_0, q_1, q_2, q_3, \ldots)$.

- $\phi_1 \mathbf{U} \phi_2$: Read as "ϕ_1 until ϕ_2". A trace satisfies $\phi_1 \mathbf{U} \phi_2$ if some suffix of the trace satisfies ϕ_2, and all other suffixes that start before it satisfy ϕ_1. Formally, a trace satisfies $\phi_1 \mathbf{U} \phi_2$ if and only if for some $j \geq 0$, the suffix $(q_j, q_{j+1}, \ldots)$ satisfies ϕ_2, and for all i, $0 \leq i < j$, the suffix $(q_i, q_{i+1}, \ldots)$ satisfies ϕ_1.

Similar to first-order logic, the semantics of LTL denote the set of traces that satisfy the LTL formula.

Example 2.11. *Consider the logical AND gate in Figure 2.1(a) as an example. The trace* $((a = false, b = false, c = false), (a = false, b = true, c = false), (a = true, b = true, c = true))$ *satisfies the LTL formula* $\mathbf{F}(c)$ *because, starting from* $j = 2$*, the suffix* $((a = true, b = true, c = true))$ *satisfies* $c = true$*.*

However, the trace does not satisfy the LTL formula $\mathbf{G}(c)$ *because the suffix* $((a = false, b = false, c = false), (a = false, b = true, c = false), (a = true, b = true, c = true))$ *does not have* c *as true in its first element.*

Since the logical AND relationship between the ports always holds, all possible behaviors or traces of the logical AND gate can be represented by $\mathbf{G}(c \Leftrightarrow a \wedge b)$*.*

2.1.4 Components

So far, we have discussed the universe of behaviors for the ports and systems, as well as the properties that define the system's requirements. We now turn to the characterization of systems by describing their interactions with the environment, referred to as *components*, which impose restrictions on the universe of behaviors. The restriction defines the behaviors the system can exhibit within the universe of behaviors. This enables us to reason and operate on the model by considering the interaction in terms of these behaviors.

Definition 2.6. *A component, denoted by M, is defined as a set of behaviors $M \subseteq \mathcal{B}_{\mathcal{P}}$.*

From the definition, there is no distinction between a property and a component in terms of their structure, as both are defined by sets. The difference lies in their usage. A component refers to the behaviors a system can exhibit, while a property denotes the requirements that a system must satisfy. Therefore, we can apply the introduced property expression, such as relational interfaces and LTL, to components, as long as the property fully captures all of their behaviors.

We use the example in Figure 2.1 to illustrate the concepts of a component and a property:

Example 2.12. *The logical AND gate in Figure 2.1(a) is a component with the following static behaviors:*

$$M_{ANDgate} = (c \Leftrightarrow (a \wedge b)).$$

The expression in first-order logic is equivalent to $\{(a, b, c) \in \mathbb{B}^3 \mid c \Leftrightarrow (a \wedge b)\}$, where the context for $(a, b, c) \in \mathbb{B}^3$ is clear.

The resistor in Figure 2.1(b), with resistance $r = 4$, is a component exhibiting the following static behaviors based on Ohm's law:

$$M_{resistor} = (V = 4I).$$

Behavior projection provides alternative views of behaviors or components by either ignoring certain ports or incorporating additional ones. Let $e \in \mathcal{B}_{\mathcal{P}}$ represent a behavior, and let $p \in \mathcal{P}$ be a port. We use e_p to denote the behavior restricted to port p.

Now, consider two port sets, $\mathcal{P}$ and $\mathcal{P}'$, and let $B_{\mathcal{P}'} \subseteq \mathcal{B}_{\mathcal{P}'}$ represent a set of behaviors defined on $\mathcal{P}'$. The projection of the set of behaviors $B_{\mathcal{P}'}$ onto ports $\mathcal{P}$ is defined as:

$$\pi_{\mathcal{P}}(B_{\mathcal{P}'}) = \left\{ e \in \mathcal{B}_{\mathcal{P}} \;\middle|\; \begin{array}{l} \exists e' \in B_{\mathcal{P}'} \ (\forall p \in \mathcal{P} \cap \mathcal{P}', \ e_p = {e'}_p) \ \wedge \\ (\forall p \in \mathcal{P} \setminus \mathcal{P}', \ {e'}_p \in \mathcal{B}_p) \end{array} \right\}.$$

The projection excludes ports not in the new view, ensures consistency of values on common ports, and extends the result by incorporating the universe of behaviors for the new ports.

Example 2.13. *This example demonstrates the projection of behaviors onto three distinct port sets derived from the system in Example 2.12.*

1. *Projection onto $\mathcal{P}_1 = \{a, b, c, d\}$ is as follows:*

$$\pi_{\mathcal{P}_1}(M_{ANDgate}) = \left\{ (a, b, c, d) \in \mathbb{B}^4 \;\middle|\; c \Leftrightarrow (a \wedge b) \right\}.$$

The new view introduces an additional port d of type $\mathbb{B}$, while preserving the existing ports. Since port d is not related to the component, it can be seamlessly incorporated by expanding the set's domain.

2. *Projection onto $\mathcal{P}_2 = \{b, c\}$ is as follows:*

$$\pi_{\mathcal{P}_2}(M_{ANDgate}) = \left\{(b, c) \in \mathbb{B}^2 \;\middle|\; \neg c \vee b\right\}.$$

The new view excludes port a from the component's behavior. We obtain this result by first enumerating all behaviors in $M_{ANDgate}$: $\{(a = false, b = false, c = false), (a = false, b = true, c = false), (a = true, b = false, c = false), (a = true, b = true, c = true)\}$. Excluding the values for a, we get:

$$\{(b = false, c = false), (b = true, c = false), (b = true, c = true)\}.$$

These behaviors are then represented by the first-order logic formula $\neg c \vee b$.

3. *Projection onto $\mathcal{P}_3 = \{b, c, d\}$ is as follows:*

$$\pi_{\mathcal{P}_3}(M_{ANDgate}) = \left\{(b, c, d) \in \mathbb{B}^3 \;\middle|\; \neg c \vee b\right\}.$$

The new view introduces an additional port d while excluding port a. This result is obtained by extending the projection from $\mathcal{P}_2$ to include the new port d, which is independent of the component's behavior.

In systems, components are connected and interact with each other through ports, where connected ports must exhibit the same values. The collection of connected components forms a new component, referred to as the *composition* of the original components:

Definition 2.7. *Given components M_1 with ports $\mathcal{P}_1$ and M_2 with ports $\mathcal{P}_2$, where $\mathcal{P}_{12} = \mathcal{P}_1 \cup \mathcal{P}_2$, the composition of M_1 and M_2, denoted $M_1 \parallel M_2$, is defined as:*

$$M_1 \parallel M_2 = \pi_{\mathcal{P}_{12}}(M_1) \cap \pi_{\mathcal{P}_{12}}(M_2).$$

The intersection is understood as the simultaneous enforcement of the restrictions imposed by the components. The projection ensures that the two components share the same universe of behaviors, making the set intersection meaningful.

Example 2.14. *Figure 2.2 depicts two logical AND gates, g_1 and g_2, and their composition. To obtain the component expression for the composition, the ports are first defined as $\mathcal{P}_1 = a, b, c$, $\mathcal{P}_2 = c, d, e$, and $\mathcal{P}_{12} = a, b, c, d, e$. The components can then be written as:*

$$M_{g_1} = \left\{(a, b, c) \in \mathbb{B}^3 \;\middle|\; c \Leftrightarrow (a \wedge b)\right\},$$
$$M_{g_2} = \left\{(c, d, e) \in \mathbb{B}^3 \;\middle|\; e \Leftrightarrow (c \wedge d)\right\}.$$

To ensure that both components share the same universe of behaviors, they are projected onto $\mathcal{P}_{12}$:

$$\pi_{\mathcal{P}_{12}}(M_{g_1}) = \left\{(a, b, c, d, e) \in \mathbb{B}^5 \;\middle|\; c \Leftrightarrow (a \wedge b)\right\},$$
$$\pi_{\mathcal{P}_{12}}(M_{g_2}) = \left\{(a, b, c, d, e) \in \mathbb{B}^5 \;\middle|\; e \Leftrightarrow (c \wedge d)\right\}.$$

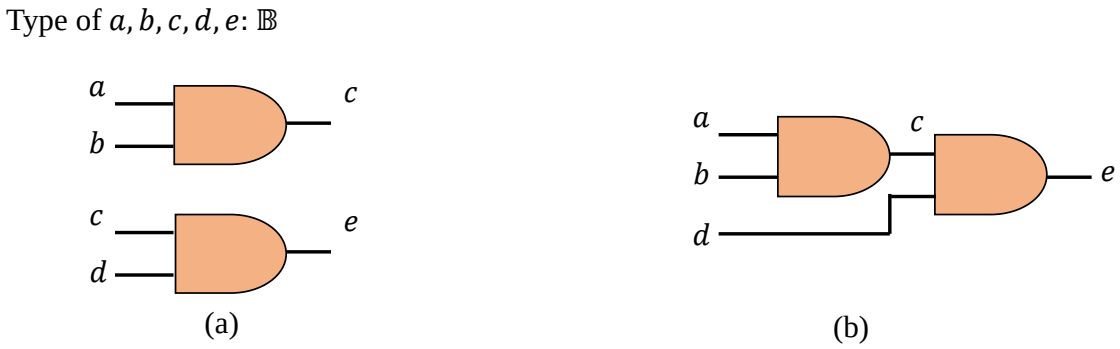


Figure 2.2: Examples of component composition: (a) two systems with their port definitions and (b) the composition result of the two systems.

Consequently, the intersection is applied to derive the composed component:

$$M_{g_1} \parallel M_{g_2} = \left\{(a,b,c,d,e) \in \mathbb{B}^5 \;\middle|\; (c \Leftrightarrow a \wedge b) \wedge (e \Leftrightarrow c \wedge d)\right\}.$$

A component satisfies a property when all behaviors it can exhibit are contained within the set of behaviors specified by the property:

Definition 2.8. *Let M be a component and P a property. M satisfies P, denoted $M \models P$, if M is a subset of P, i.e., $M \subseteq P$.*

Example 2.15. *The component $M_{ANDgate}$ in Example 2.12(a) satisfies the property $P = (c \Rightarrow b)$. This can be verified by enumerating all elements in the set $M_{ANDgate}$ and observing that they are all elements of P. Therefore, $M_{ANDgate}$ satisfies the property P.*

2.1.4.1 Tools for System Reasoning: Model Checking and SMT

Since properties and components may be defined using different formalisms, techniques known as *model checking* have been developed to verify whether a property is satisfied by a component within these formalisms. For example, model-checking can be used to verify if a finite-state machine satisfies a property specified in temporal logic. Note that a component can also be used as a property to check if component M_1 always produces behaviors available from M_2, or equivalently, if $M_1 \subseteq M_2$.

Given the compact encoding of sets, a question arises: how can the relationship between sets be reasoned efficiently? For first-order logic and LTL, set operations can be performed through binary operations: A binary AND on two formulas represents the intersection of the sets, a binary OR on two formulas creates the union of the sets, and negation represents the complement of a set. However, it is also necessary to query elements within a set, check whether a set is empty, and verify the subset relationship between sets. These reasoning

tasks require an additional technique beyond binary operations: satisfiability modulo theories (SMT) solving.

An SMT problem asks whether a formula, with background theories that interpret certain predicates and function symbols [12], can be evaluated to true (satisfied) for some assignment to the variables and functions. The following is an example of an SMT formula in the nonlinear arithmetic background theory:

$$x * x \leq y \wedge \neg(y \leq x + 1),$$

where x and y are variables interpreted as real numbers, 1 is the multiplicative identity, and the symbols $*$, $+$, $\leq$ represent the usual operations on the set of real numbers $\mathbb{R}$. An SMT problem is said to be satisfiable if there exists an assignment to the variables that satisfies the formula. For example, the formula above is satisfiable, as the assignment $x = 1$ and $y = 3$ satisfies it.

Reasoning about sets can be converted into SMT problems. Querying a behavior from the set represented by a formula ϕ involves solving the SMT problem for ϕ and obtaining the corresponding assignment. A set is empty if the SMT problem for ϕ is unsatisfiable, meaning no assignment can make the formula evaluate to true. To check whether a behavior set A, represented by ϕ_A, is a subset of a behavior set B, represented by ϕ_B, the formula $\phi_A \wedge \neg\phi_B$ is checked for satisfiability. If $\phi_A \wedge \neg\phi_B$ is satisfiable, the satisfying assignment provides a counterexample where a behavior in A is not in B. Conversely, if it is unsatisfiable, no such counterexample exists, proving that $A \subseteq B$.

While the SMT problem is undecidable for most background theories, it has been shown to be applicable in numerous real-world scenarios, including formal verification, synthesis, and scheduling. Various tools for SMT solving have been developed, such as Z3 [48], CVC4 [13], and MathSAT [36]. In addition, model checkers such as Spin [69], TLA+ [93], nuXmv [28], and UCLID5 [157] offer support for property verification, with the latter two integrating SMT solving to allow more expressive descriptions of properties and systems.

2.2 Contracts

This section introduces contracts, the contract-based design methodology, and the assume-guarantee contract, one of the most widely used contract formalisms due to its compact encoding and ease of use.

2.2.1 The Meta-theory of Contracts

A contract is a formal specification for a system, defined as a pair of component sets $\mathcal{C} = (\mathcal{E}, \mathcal{I})$. The environment set $\mathcal{E}$ includes components that can act as the system's environment to ensure normal operation, while the implementation set $\mathcal{I}$ consists of components capable of realizing the specification. A contract is considered *consistent* if $\mathcal{I}$ is non-empty, meaning

it has at least one implementation, and *compatible* if $\mathcal{E}$ is non-empty, indicating the existence of at least one environment.

The above definition is abstract and forms the foundation of the *meta-theory of contracts* [17]. It encompasses various concrete formalisms, such as assume-guarantee contracts [20], rely-guarantee reasoning [83], and interface theories [6]. Examples of contracts will be presented in the discussion of assume-guarantee contracts, which is introduced later in this chapter. For a detailed discussion of the meta-theory of contracts, see the monograph by Benveniste *et al.* [17]; for algebraic properties based on this theory, refer to Inigo's work [77].

As mentioned earlier, decomposing formal specifications can significantly facilitate the design process. To support this, the notions of *refinement* and *composition* are introduced for contracts.

Refinement and Abstraction *Refinement* and *abstraction* describe relationships between contracts that determine whether one contract can safely replace another without violating its requirements.

Definition 2.9. *Let $\mathcal{C} = (\mathcal{E},\mathcal{I})$ and $\mathcal{C}' = (\mathcal{E}',\mathcal{I}')$ be two contracts. The contract $\mathcal{C}'$ is a refinement of $\mathcal{C}$, or equivalently, $\mathcal{C}$ is an abstraction of $\mathcal{C}'$, denoted as $\mathcal{C} \succeq \mathcal{C}'$ or $\mathcal{C}' \preceq \mathcal{C}$, if and only if the following conditions hold:*

$$\mathcal{E}' \supseteq \mathcal{E},$$
$$\mathcal{I}' \subseteq \mathcal{I}.$$

Intuitively, the refined contract $\mathcal{C}'$ can serve as a new specification for developing the system originally specified by $\mathcal{C}$, as it can operate in all environments within $\mathcal{E}$, and its implementations are necessarily included in $\mathcal{I}$.

Refinement induces a partial ordering over contracts. Contract theories also require the existence of a *shared refinement* between any two contracts $\mathcal{C}_1$ and $\mathcal{C}_2$:

Definition 2.10. *The shared refinement, or greatest lower bound (GLB), of two contracts $\mathcal{C}_1$ and $\mathcal{C}_2$ is a contract, denoted by $\mathcal{C}_1 \wedge \mathcal{C}_2$, satisfying $\mathcal{C}_1 \wedge \mathcal{C}_2 \preceq \mathcal{C}_1$ and $\mathcal{C}_1 \wedge \mathcal{C}_2 \preceq \mathcal{C}_2$, where the operator $\wedge$ is referred to as the conjunction operator.*

The shared refinement can be intuitively understood as generating a single contract that captures the requirements of both contracts, with each contract specifying different conditions for the same component.

Composition Composition, the inverse of decomposition, generates the overall contract of a system from the contracts of its subsystems, assuming the connections between these subsystems have already been encoded in the formalism.

Definition 2.11. *The composition of two contracts* $\mathcal{C}_1 = (\mathcal{E}_1, \mathcal{I}_1)$ *and* $\mathcal{C}_2 = (\mathcal{E}_2, \mathcal{I}_2)$*, denoted as* $\mathcal{C}_1 \parallel \mathcal{C}_2$*, is defined as the minimum contract (with respect to refinement)* $\mathcal{C}_{comp} = \mathcal{C}_1 \parallel \mathcal{C}_2 = (\mathcal{E}_{comp}, \mathcal{I}_{comp})$*, such that for every* $M_1 \in \mathcal{I}_1$*,* $M_2 \in \mathcal{I}_2$*, and* $E \in \mathcal{E}_{comp}$*, the following conditions hold:*

$$M_1 \parallel M_2 \in \mathcal{I}_{comp},$$
$$E \parallel M_1 \in \mathcal{E}_2,$$
$$E \parallel M_2 \in \mathcal{E}_1.$$

Composition can be intuitively understood as defining the system's requirements based on those of the subsystems. Any system implementation must be the composition of the subsystems' implementations, while the system environments must be those that, when interacting with any subsystem implementation, can create a valid environment for the other subsystem. This ensures that the system environment satisfies the requirements of both subsystems. Using the composition notion, the contracts $\mathcal{C}_1$ and $\mathcal{C}_2$ decompose the contract $\mathcal{C}_s$ if $\mathcal{C}_1 \parallel \mathcal{C}_2 \preceq \mathcal{C}_s$.

The composition introduces an important property: *independent refinement*.

Property 2.1. *Given contracts* $\mathcal{C}_1$*,* $\mathcal{C}_2$*,* $\mathcal{C}_1'$*,* $\mathcal{C}_2'$*, if* $\mathcal{C}_1' \preceq \mathcal{C}_1$ *and* $\mathcal{C}_2' \preceq \mathcal{C}_2$*, then:*

$$\mathcal{C}_1' \parallel \mathcal{C}_2' \preceq \mathcal{C}_1 \parallel \mathcal{C}_2.$$

This property shows that contracts $\mathcal{C}_1$ and $\mathcal{C}_2$ can be refined independently while preserving the overall refinement relationship of their composition. Consequently, the independent development of subsystems satisfying $\mathcal{C}_1'$ and $\mathcal{C}_2'$ can be integrated to form the implementation of $\mathcal{C}_1 \parallel \mathcal{C}_2$. This supports the top-down design process shown in Figure 1.2, where contracts are first decomposed, and then the implementations for the decomposed subsystems are developed to satisfy the overall requirement.

Refinement, abstraction, and composition involve reasoning about contracts by manipulating their corresponding sets. Together, these are referred to as *contract manipulations*.

2.2.2 Contract-based Design Methodology

This part revisits the contract-based design methodology [151] introduced in Chapter 1.3.1, building on the previously defined concepts and properties of contracts.

Contract-based design adopts a divide-and-conquer approach to address the limitations of the V-model while preserving its advantages. In a typical contract-based design process, all design requirements and elements are represented as contracts, which formally define the requirements and characterize the behavior and expected environment of each element. This methodology leverages refinement to ensure that design requirements are met, and uses composition and decomposition to progressively break down the design problem and guide the development process.

Later chapters provide more details on refinement and how design goals are satisfied. For now, the focus is on decomposing contracts to refine the original specifications.

The adoption of contract-based design can follow a bottom-up, top-down, or hybrid approach, with the hybrid approach demonstrating a meet-in-the-middle strategy, as in platform-based design [87]. From a bottom-up perspective, contracts can be used for verification at the integration stage of the V-shaped design process. In this approach, the development results are characterized as contracts and verified against the design requirements. Verification is performed through refinement, which involves checking whether the contract representing a subsystem's development result is a refinement of its corresponding subsystem contract, and whether the composition of subsystem contracts refines the system-level requirements. Although primarily used for verification, this approach also supports synthesis, enabling a correct-by-construction approach. Specifically, contracts can guide the selection of existing design elements that refine the given contracts, allowing these elements to be integrated as subsystems.

In addition to the bottom-up apporach, contract-based design can incorporate a top-down approach. In this approach, the system's top-level contracts are decomposed into subsystem contracts, as illustrated in Figure 1.2. The composition of the decomposed contracts must refine the top-level system contracts to ensure the design requirements are satisfied. The independent refinement property enables each decomposed contract to be developed independently. By breaking the design problem into smaller and more manageable independent subsystem design problems, the overall design complexity can be reduced.

In the meet-in-the-middle style as in platform-based design, the bottom-up approach forms a platform by abstracting and composing contracts that characterize the available design elements. The abstraction is defined by the notion of vertical contracts [120], which specify how to transition between levels of abstraction within the platform and expose underlying details as the design process progresses downward. Through this approach, the top-down approach converges with the bottom-up approach at the platform, enabling the decomposition of contracts based on evaluations and estimations within the platform. This iterative process continues until the design is mapped to contracts that represent the underlying design elements.

In addition to supporting different design approaches, contract-based design facilitates requirement management and complexity reduction by enabling the separate specification of different design aspects through viewpoints [20, 132]. For example, the conjunction operator can combine various conditions specified for the same design, resulting in a shared refinement that integrates these conditions into a single contract.

With its various design approaches, the contract-based design methodology offers several benefits, including early integration testing, independent development, design space definition for optimization, and facilitation of component reuse. The following sections discuss these advantages in detail.

2.2.2.1 Early Integration Testing

Integration testing verifies whether the integration of subsystem implementations satisfies the design requirements. Early integration testing shifts some of these checks to earlier stages, enabling verification before the design is physically implemented. Since contracts define the formal requirements of a system, they can be used throughout the design process to verify correctness at each stage, thereby enable early integration testing. The composition of subsystem contracts represents the expected integration result, assuming subsequent development proceeds correctly. Refinement can then be used to check whether this result satisfies the top-level contract. If the refinement holds, it guarantees that the integration of all possible implementations will meet the top-level specification. This capability is valuable in all contract-based design approaches. In the bottom-up approach, it helps determine whether selected subsystem designs collectively satisfy system-level requirements. In the top-down approach, early integration testing is implicitly performed during contract decomposition, since the refinement relation must hold between the composition of the decomposed contracts and the original contract.

Additionally, because contracts are defined in terms of environment and implementation behaviors, they support the prediction and analysis of system behavior before physical integration. By capturing these behaviors, contracts enable early performance estimation and assessment, which is especially important for optimization.

Early integration testing is crucial in applications where the cost of design faults is high and the consequences are severe, such as in space missions and aircraft design. In these cases, a single error can lead to significant revenue loss, extended development cycles, or even casualties. Since contracts enable integration testing before the actual system is implemented, they help identify specification errors and estimate design performance early in the process. This allows designers to address issues sooner, saving both time and money while reducing the risk of failures in the final implementation. Once manufacturing begins, subsystem implementations only need to satisfy their corresponding contracts, with the correctness of the entire integration ensured through early integration testing and contract theory.

2.2.2.2 Independent Design

The independent refinement property in Property 2.1 allows each subsystem contract to be independently refined and developed, without violating the refinement relation of the composition result. The decomposed subsystem contracts can thus be assigned to separate design teams. The refinement relation does not need to be verified again as long as each design team guarantees that their development refines their assigned contracts.

We refer to this benefit of contract-based design as the *independent design* paradigm. Independent design is especially valuable in supporting the role of original equipment manufacturers (OEMs), also known as suppliers, who produce parts for integration into systems designed by other companies. Companies can delegate subsystem design tasks to OEMs by

providing contracts. Each OEM can independently develop its part according to the assigned contract, without access to system-level specifications or coordination with other suppliers. This paradigm enables early detection of design faults at the specification stage, reducing the risk of costly and time-consuming redesigns. It also helps protect high-level design concepts from being disclosed to suppliers, who may belong to different organizations.

Independent design reduces problem size by eliminating dependencies between subsystem design problems. It also enables parallel development without waiting for other design teams. Additionally, it clarifies responsibilities in the OEM supply chain and requirement management. OEMs are only responsible for delivering parts that satisfy the assigned contracts, while the responsibility for integration lies with the company that derived these contracts.

2.2.2.3 Defining Design Space for Optimization

Another benefit of contract-based design is its clear, formal definition of the design space for optimization. Rather than relying on vague specifications—which depend on designer heuristics and prior experience to interpret, often leading to limited design space exploration—contracts explicitly define the available implementations and expected environments, resulting in a well-defined design space for thorough exploration.

The implementations specified by contracts define the available choices for satisfying the contract, thus forming the design space for a design problem. The environment in a contract specifies the conditions under which the system is expected to operate normally. In other words, it identifies conditions outside of normal operation. The system's behavior under these conditions does not affect requirement satisfaction and can therefore be leveraged as flexibility in performance-related design decisions. For example, in digital circuit logic optimization, the "don't care" terms can be used when certain input values are known to never occur. In such cases, the logic function's output under these inputs can be assigned based on their impact on circuit size, as they do not affect the satisfaction of design requirements. Similarly, by specifying the environment in contracts, designers can identify conditions that do not affect requirement satisfaction. The corresponding behaviors under those conditions can then be determined based on other design considerations, such as cost, timing, and power.

2.2.2.4 Component Reuse

Contract-based design also facilitates component reuse, allowing existing development results from other designs to be applied without starting from scratch. These results can be organized into a component library, with each entry characterized by its contract. During the design process, if an existing result is found to satisfy a subsystem contract, it can be reused directly, enabling efficient integration and reducing redundant work.

Component reuse reduces both design time and cost, helping to meet time-to-market constraints and lower overall development expenses. When a suitable component is found in the library, it can be directly adopted, avoiding the need to repeat the entire design process

and enabling faster product development. Moreover, if a component is broadly applicable across multiple designs, mass production can further reduce manufacturing costs.

While full optimization should ultimately be applied to a mature design to maximize performance and minimize long-term costs, time-to-market constraints and the need to reduce initial expenses are also critical considerations. A similar concept can be seen in the common industry practice of using field-programmable gate arrays (FPGA) or existing intellectual properties (IP) for prototyping, followed by the development of in-house application-specific integrated circuits (ASIC). Identifying and leveraging existing working solutions can significantly reduce early development effort and accelerate product launch in competitive markets.

2.3 Assume-Guarantee Contracts

Building on the concept of contracts and contract-based design, this section introduces *assume-guarantee contracts* [20], a contract formalism used throughout this dissertation. Assume-guarantee contracts are widely adopted due to their compact representation and ease of use.

Definition 2.12. *An assume-guarantee contract, denoted by $\mathcal{C}^{ag}$, consists of a pair of behavior sets (A, G), where A represents the assumption set and G represents the guarantee set. Both A and G are subsets of the universe of system behaviors $\mathcal{B}_{\mathcal{P}}$.*

The assumption and guarantee sets are defined by the behaviors over the system ports $\mathcal{P}$. The assumption set describes the property of the targeted environments in which the system is expected to operate normally. In the context of the meta-theory of contracts, the environment set of an assume-guarantee contract consists of all components that satisfy the property A: $\mathcal{E} = \{E \subseteq \mathcal{B}_{\mathcal{P}} \mid E \models A\}$. Therefore, the environment set of an assume-guarantee contract is $\mathcal{E} = 2^A$, where 2^A denotes the power set (i.e., the set of all subsets) of A.

The guarantee set defines the property that the design must satisfy when operating within the targeted environment. Accordingly, an implementation must satisfy the property for every environment in the environment set: $\mathcal{I} = \{M \subseteq \mathcal{B}_{\mathcal{P}} \mid \forall E \in \mathcal{E},\ M \parallel E \models G\}$. As a result, the implementation set can be expressed as $\mathcal{I} = 2^{G \cup \overline{A}}$, where $\overline{A}$ denotes the complement of A in the universe $\mathcal{B}_{\mathcal{P}}$, i.e., $\overline{A} = \mathcal{B}_{\mathcal{P}} \setminus A$. In this way, assume-guarantee contracts are instantiated as contracts.

Note that $G \cup \overline{A}$ represents the set of behaviors that are acceptable for the system to produce. When the system operates outside its targeted environments, its behavior is unconstrained, and any behavior is considered acceptable. As a result, the acceptable behavior set of a contract $\mathcal{C}^{ag} = (A, G)$ is $G \cup \overline{A}$, and an implementation must be a subset of this set. Formally, the implementation set can be written as: $\mathcal{I} = \left\{M \subseteq \mathcal{B}_{\mathcal{P}} \;\middle|\; M \subseteq G \cup \overline{A}\right\}$, which explains why $\mathcal{I} = 2^{G \cup \overline{A}}$.

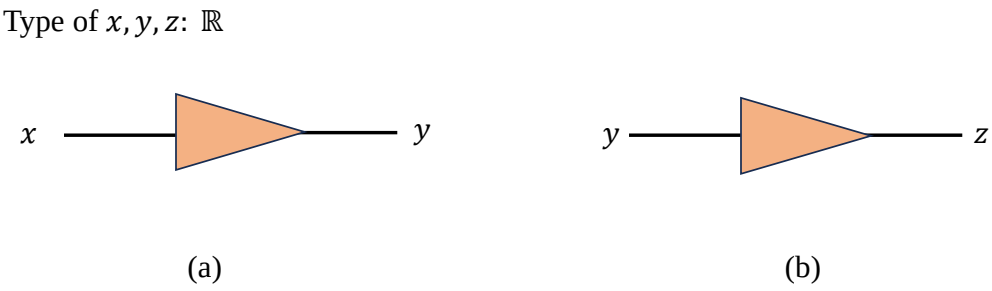


Figure 2.3: Examples of systems to be expressed as contracts, showing ports, port types, and the corresponding system diagram.

Assume-guarantee contracts can be directly related to contracts in the meta-theory through the concept of *saturation*, based on the definitions of acceptable behaviors and implementation sets:

Definition 2.13. *An assume-guarantee contract $\mathcal{C}^{ag} = (A, G)$ is said to be saturated if it satisfies $G = G \cup \overline{A}$.*

A saturated contract ensures that the guarantee set includes all acceptable behaviors. Any assume-guarantee contract can be saturated using the saturation operator:

$$\text{sat}_{\text{ag}}(A, G) = (A, G \cup \overline{A}).$$

Saturation makes both the environment and implementation sets explicit: the power set of the assumption defines the environment set, and that of the guarantee defines the implementation set. The semantics of the specification remains unchanged, as saturation preserves the set of environments and implementations.

Example 2.16. *An assume-guarantee contract $\mathcal{C}^{ag} = (A, G) = (x \geq 0, y = 2x)$ characterizes an amplifier with input port x and output port y, as shown in Figure 2.3(a). The assumption specifies that the input must be greater than or equal to 0 for the amplifier to function, defining the environment as any component that satisfies $x \geq 0$. The guarantee ensures that, when the amplifier operates within these environments, the output is double the input value.*

The contract can be saturated by replacing $G = (y = 2x)$ with $G \cup \overline{A} = (y = 2x) \vee (x < 0)$, resulting in $\mathcal{C}^{ag}_{sat} = sat_{ag}(A, G) = (x \geq 0, (y = 2x) \vee (x < 0))$.

In addition to characterizing a system, a contract can serve as a design requirement, with the guarantee offering flexibility in the implementation:

Example 2.17. *An assume-guarantee contract $\mathcal{C}^{ag} = (A, G) = (x \geq 0, (y \geq 1.9x) \wedge (y \leq 2.1x))$ can serve as a formal specification for the amplifier shown in Figure 2.3(a). The*

key difference is that the guarantee now specifies a range, meaning an implementation must produce an output that falls within this range for all environments satisfying $x \geq 0$. This range introduces flexibility, as any output within the specified bounds satisfies the contract.

Similarly, the contract can be saturated by replacing $G = (y \geq 1.9x) \wedge (y \leq 2.1x)$ with $G \cup \overline{A} = ((y \geq 1.9x) \wedge (y \leq 2.1x)) \vee (x < 0)$, resulting in $\mathcal{C}^{ag}_{sat} = sat_{ag}(A, G) = (x \geq 0, ((y \geq 1.9x) \wedge (y \leq 2.1x)) \vee (x < 0))$.

In the following sections, we introduce manipulations of assume-guarantee contracts, including their properties, operations, and relations used for contract reasoning.

2.3.1 Contract Properties

In the previous part, saturation was introduced as a property of assume-guarantee contracts, indicating whether a contract is saturated or unsaturated. In this part, we introduce several additional properties derived from the definition of assume-guarantee contracts.

2.3.1.1 Obligation

The acceptable behavior set $G \cup \overline{A}$ includes behaviors that will never occur under the targeted environments, due to the inclusion of $\overline{A}$. To describe the behaviors exhibited when the system operates within the targeted environments, the notion of *obligation* [20] is introduced:

Definition 2.14. *The obligation of a contract $\mathcal{C}^{ag} = (A, G)$ is the behavior set $A \cap G$.*

The obligation represents the set of acceptable behaviors under the targeted environments. If the obligation of a contract is empty, the contract is said to be *vacuous*, indicating that no implementation can exhibit any valid behavior, even when operating in any targeted environment. An implementation that cannot exhibit any valid behaviors under a targeted environment is referred to as a *vacuous implementation.*

Example 2.18. *The obligation of the contract $\mathcal{C}^{ag} = (A, G) = (x \geq 0, y = 2x)$ in Example 2.16 is $(x \geq 0) \wedge (y = 2x)$, which includes all pairs (x, y) such that x is greater than or equal to 0 and y equals twice x.*

2.3.1.2 Consistency

This property directly follows from the meta-theory [17]:

Definition 2.15. *A contract is considered consistent if its implementation set is not an empty set, i.e., $G \cup \overline{A} \neq \emptyset$.*

Example 2.19. *The contract in Example 2.16 is consistent, as $G \cup \overline{A} = (y = 2x) \vee (x < 0)$ is non-empty. On the other hand, the contract $\mathcal{C}^{ag} = (\mathcal{B}_{\mathcal{P}}, \emptyset)$ is inconsistent.*

2.3.1.3 Compatibility

Similar to consistency, this property follows from the meta-theory [17]:

Definition 2.16. *A contract is considered compatible if its environment set is not an empty set, i.e.,* $A \neq \emptyset$.

Example 2.20. *The contract in Example 2.16 is compatible, as its environment set* $x \geq 0$ *is non-empty. On the other hand, the contract* $\mathcal{C}^{ag} = (\emptyset, y = 2x)$ *is incompatible.*

From the definition, it follows that if an assume-guarantee contract is compatible and its assumption is not the universal behavior set, then the contract must be consistent. A contract is inconsistent only when the assumption equals the universal behavior set and the guarantee is empty, meaning the design does not exhibit any behaviors.

2.3.2 Relations between Assume-guarantee Contracts

This part introduces important relations between assume-guarantee contracts, including the refinement relation and two others: *conformance* and *strong dominance*.

2.3.2.1 Refinement

Using the implementation and environment sets of assume-guarantee contracts, the refinement relation can be defined as follows:

Definition 2.17. *Given two assume-guarantee contracts* $\mathcal{C}_1^{ag} = (A_1, G_1)$ *and* $\mathcal{C}_2^{ag} = (A_2, G_2)$, $\mathcal{C}_1^{ag} \succeq \mathcal{C}_2^{ag}$ *holds when the following conditions are satisfied:*

$$A_1 \subseteq A_2,$$
$$(G_1 \cup \overline{A_1}) \supseteq (G_2 \cup \overline{A_2}).$$

The first condition requires that the assumption set of the refined contract be more relaxed (i.e., larger in the subset relation) than that of the abstract contract, indicating its ability to operate in at least all environments specified by the abstract contract. The second condition requires that the implementation set of the refined contract be more stringent than that of the abstract contract, ensuring that any implementation of the refined contract is also a valid implementation of the abstract contract.

Example 2.21. *Consider the following contracts for the system shown in Figure 2.3(a):*

$$\mathcal{C}_1^{ag} = (A, G) = (x \geq 0, (y \geq 1.9x) \wedge (y \leq 2.1x)),$$
$$\mathcal{C}_2^{ag} = (A, G) = (x \geq -1, y = 2x).$$

To verify that $\mathcal{C}_1^{ag} \succeq \mathcal{C}_2^{ag}$, *first observe that* $A_1 = (x \geq 0) \subseteq (x \geq -1) = A_2$, *showing that the environment of the refined contract is contained within the environment of the abstract*

contract. Next, for the implementation, $G_1 = ((y \geq 1.9x) \wedge (y \leq 2.1x)) \supseteq (y = 2x) = G_2$. *Therefore,* $G_1 \cup \overline{A_1} \supseteq G_2 \cup \overline{A_2}$, *confirming that* $\mathcal{C}_1^{ag} \succeq \mathcal{C}_2^{ag}$ *holds.*

Intuitively, if an implementation satisfies the requirements of $\mathcal{C}_2$, *it works for any input* $x \geq -1$, *which includes environments where* $x \geq 0$. *Additionally, the property* $y = 2x$ *always satisfies* $((y \geq 1.9x) \wedge (y \leq 2.1x))$, *ensuring that the requirement specified by* $\mathcal{C}_2$ *is never violated.*

2.3.2.2 Conformance

Conformance means that the obligation of one contract is contained within the obligation of another contract [20]:

Definition 2.18. *Given two assume-guarantee contracts* $\mathcal{C}_1^{ag} = (A_1, G_1)$ *and* $\mathcal{C}_2^{ag} = (A_2, G_2)$, $\mathcal{C}_2^{ag}$ *conforms to* $\mathcal{C}_1^{ag}$ *if the following condition holds:*

$$A_2 \cap G_2 \subseteq A_1 \cap G_1.$$

2.3.2.3 Strong Dominance

Strong dominance combines the concepts of refinement and conformance to define a more restrictive relation between contracts [20]:

Definition 2.19. *Given two assume-guarantee contracts* $\mathcal{C}_1^{ag} = (A_1, G_1)$ *and* $\mathcal{C}_2^{ag} = (A_2, G_2)$, $\mathcal{C}_2^{ag}$ *strongly dominates* $\mathcal{C}_1^{ag}$ *if* $\mathcal{C}_2^{ag}$ *conforms to* $\mathcal{C}_1^{ag}$ *and* $\mathcal{C}_1^{ag} \succeq \mathcal{C}_2^{ag}$.

2.3.3 Operations of Assume-guarantee Contracts

In the meta-theory, *composition* generates the overall contract of a system from the contracts of its subsystems, enabling reasoning about the system's requirements based on its subsystems *Conjunction*, in a similar fashion but with a different meaning, produces a contract that refines multiple contracts to identify the overall requirement based on different scenarios. These operations are essential for reasoning about system requirements during the design process. This part introduces the operations of assume-guarantee contracts to facilitate reasoning about system requirements. Table 2.1 summarizes these operations. Note that these operations are defined over saturated contracts, as saturated contracts exhibit favorable algebraic properties [77].

2.3.3.1 Composition and Quotient

Definition 2.20. *The composition of two saturated contracts* $\mathcal{C}_1^{ag} = (A_1, G_1)$ *and* $\mathcal{C}_2^{ag} = (A_2, G_2)$, *denoted by* $\mathcal{C}_1^{ag} \parallel \mathcal{C}_2^{ag}$, *is computed as*

$$\mathcal{C}_1^{ag} \parallel \mathcal{C}_2^{ag} = ((A_1 \cap A_2) \cup \overline{(G_1 \cap G_2)}, G_1 \cap G_2).$$

Operation Name	Notation	Computation
Composition	$\mathcal{C}_1^{ag} \parallel \mathcal{C}_2^{ag}$	$((A_1 \cap A_2) \cup \overline{(G_1 \cap G_2)}, G_1 \cap G_2)$
Quotient	$\mathcal{C}_1^{ag} / \mathcal{C}_2^{ag}$	$(A_1 \cap G_2, (A_2 \cap G_1) \cup \overline{(A_1 \cap G_2)})$
Conjunction	$\mathcal{C}_1^{ag} \wedge \mathcal{C}_2^{ag}$	$(A_1 \cup A_2, G_1 \cap G_2)$
Implication	$\mathcal{C}_1^{ag} \rightarrow \mathcal{C}_2^{ag}$	$((A_2 \cap \overline{A_1}) \cup (G_1 \cap \overline{G_2}), G_2 \cup \overline{G_1})$
Merging	$\mathcal{C}_1^{ag} \cdot \mathcal{C}_2^{ag}$	$(A_1 \cap A_2, (G_1 \cap G_2) \cup \overline{(A_1 \cap A_2)})$
Separation	$\mathcal{C}_1^{ag} \div \mathcal{C}_2^{ag}$	$((A_1 \cap G_2) \cup \overline{(A_2 \cap G_1)}, A_2 \cap G_1)$
Disjunction	$\mathcal{C}_1^{ag} \vee \mathcal{C}_2^{ag}$	$(A_1 \cap A_2, G_1 \cup G_2)$
Coimplication	$\mathcal{C}_1^{ag} \not\rightarrow \mathcal{C}_2^{ag}$	$(A_2 \cup \overline{A_1}, (G_2 \cap \overline{G_1}) \cup (A_1 \cap \overline{A_2}))$

Table 2.1: The summary of contract operations.

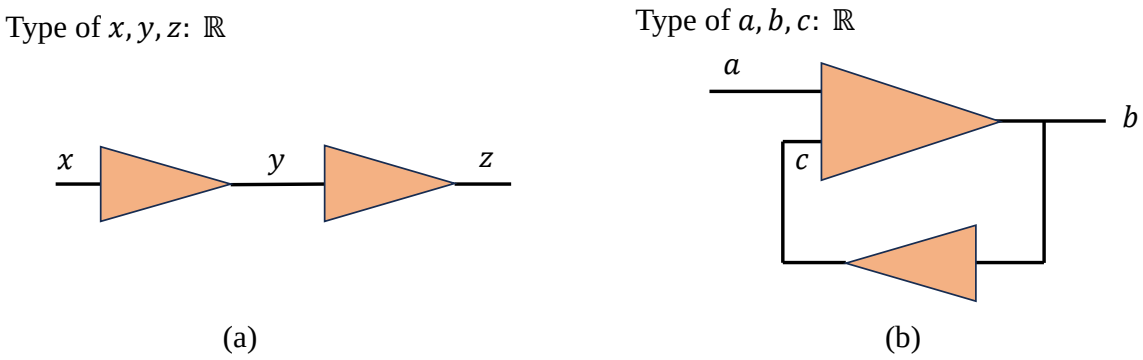


Figure 2.4: Examples of system composition: (a) cascade composition without feedback loops, and (b) feedback composition.

The composition of contracts represents the overall system specification, integrating the specifications of all subsystems. It enables reasoning at the system level, rather than focusing solely on individual subsystems. The inclusion of $\overline{(G_1 \cap G_2)}$ in the assumption set reflects that a subsystem's assumption can be satisfied by the guarantees of the other subsystem, rather than relying entirely on the external environment.

Example 2.22. *Considering the following contracts for systems in Figure 2.3(a) and 2.3(b):*

$$\mathcal{C}_1^{ag} = (A_1, G_1) = (x \geq 0, y = 2x),$$
$$\mathcal{C}_2^{ag} = (A_2, G_2) = (y \geq 2, z = 2y).$$

The contract for their composition, as illustrated in Figure 2.4, can be computed by first

saturating the individual contracts:

$$\mathcal{C}_1^{ag} = (A_1, G_1) = (x \geq 0, (y = 2x) \vee (x < 0)),$$
$$\mathcal{C}_2^{ag} = (A_2, G_2) = (y \geq 2, (z = 2y) \vee (y < 2)).$$

Then, applying the composition operation yields:

$$\mathcal{C}_{12}^{ag} = \mathcal{C}_1^{ag} \parallel \mathcal{C}_2^{ag} = (A_{12}, G_{12}),$$
$$A_{12} = ((x \geq 0) \wedge (y \geq 2) \vee (y \neq 2x \wedge x \geq 0) \vee (z \neq 2y \wedge y \geq 2)),$$
$$G_{12} = ((y = 2x \vee x < 0) \wedge (z = 2y \vee y < 2)).$$

The composition result may appear complex but can be analyzed by considering different cases based on the system input x.

- *Case 1 ($x \geq 1$):*

  When the input is $x \geq 1$, any values of y and z satisfy the assumption. If $y \geq 2$, the assumption is clearly satisfied. If $y < 2$, then $(y \neq 2x) \wedge (x \geq 0)$ must hold, since $y \neq 2x$ is guaranteed by $x \geq 1$ and $y < 2$.

  This highlights the importance of including $\overline{(G_1 \cap G_2)}$ in the assumption, as it ensures that any values of y and z satisfy the assumption, allowing these values to be determined by the subsystems without relying on the environment. The condition $x \geq 1$ ensures that the assumption A_1 is satisfied, which in turn guarantees $G_1 = (y = 2x)$ and results in $y \geq 2$, thus satisfying the assumption A_2.

  As the assumption is satisfied, the guarantee must be enforced. Given $x \geq 1$, we have $y = 2x$, which implies $y \geq 2$, and consequently, $z = 2y = 4x$.

- *Case 2 ($0 \leq x < 1$): When $y \geq 2$, the assumption is satisfied. However, there are no corresponding behaviors, as $y = 2x \vee x < 0$ evaluates to false, reflecting that the subsystem for $\mathcal{C}_1^{ag}$ cannot operate under such conditions.*

  When $y < 2 \wedge y = 2x$, the assumption is violated.

  When $y < 2 \wedge y \neq 2x$, the assumption is satisfied. However, similar to the case of $y \geq 2$, there are no corresponding behaviors.

  As a result, the system is not expected to operate normally under $0 \leq x < 1$.

- *Case 3 ($x < 0$): In this case, the contract does not exhibit any behaviors, as $(z \neq 2y \wedge y \geq 2)$ contradicts $(z = 2y \vee y < 2)$. Therefore, the system is not expected to operate normally under such input.*

Therefore, the result of the composition can be summarized as follows:

- *When the input satisfies $x \geq 1$, all subsystem assumptions are satisfied, and their guarantees are enforced.*
- *For other input values, there is either a violation of assumptions or no behaviors allowed by the contracts, indicating a failure in part of the system.*

A contract can also be composed in a feedback loop, where the port connections create a cycle.

Example 2.23. *(Feedback Composition) As shown in Figure 2.4(b), assume that the upper system is specified by $\mathcal{C}_1^{ag}$, and the lower system is specified by $\mathcal{C}_2^{ag}$, defined as follows:*

$$\mathcal{C}_1^{ag} = (A_1, G_1) = (a > 0, b = 5(a - c)),$$
$$\mathcal{C}_2^{ag} = (A_2, G_2) = (true, c = 0.1b).$$

The composition result is computed as:

$$\mathcal{C}_{12}^{ag} = \mathcal{C}_1^{ag} \parallel \mathcal{C}_2^{ag} = (A_{12}, G_{12}),$$
$$A_{12} = (a > 0 \vee (b \neq 5(a - c) \wedge a > 0) \vee c \neq 0.1b),$$
$$G_{12} = ((b = 5(a - c) \vee a \leq 0) \wedge (c = 0.1b)).$$

The obligation of the contracts requires that $b = \frac{5}{1+5\times0.1}a$ and $a > 0$, representing the closed-loop gain of the feedback amplifier.

Note that in contract composition, we use a nondeterministic semantics for the feedback behavior, as opposed to a constructive fixed-point semantics [160, 52], which seeks a fixed point that is guaranteed to be reached from unknown values in the system. This approach simplifies the operation, but it comes with the tradeoff that the stability and reachability of these fixed points are not guaranteed. As a result, it leads to an over-approximation of all possible behaviors in the system.

The operation remains the same, regardless of whether the composition involves feedback, making it easier to reason about system compositions without needing to consider the topology of the connections.

The quotient in Table 2.1 is the inverse operation of composition, also known as the *adjoint* operator for composition. Given a system contract $\mathcal{C}_{12}^{ag}$ and a subsystem contract $\mathcal{C}_1^{ag}$, it can find the requirement for the missing subsystem by computing $\mathcal{C}_{12}^{ag}/\mathcal{C}_1^{ag}$.

2.3.3.2 Conjunction and Implication

Conjunction combines the specifications of a system in different scenarios. For assume-guarantee contracts, it is defined as follows:

Definition 2.21. *The conjunction of two saturated contracts $\mathcal{C}_1^{ag} = (A_1, G_1)$ and $\mathcal{C}_2^{ag} = (A_2, G_2)$, denoted by $\mathcal{C}_1^{ag} \wedge \mathcal{C}_2^{ag}$, is computed as:*

$$(A_1 \cup A_2, G_1 \cap G_2).$$

The operation combines contracts from different scenarios, similar to the concept of the conditional expressions ("if") in programming languages. The assumption of each contract corresponds to the condition of the conditional expressions, while the guarantee defines the outcome of the corresponding branch. Note that there is no inherent order among the conditions. When an environment satisfies both A_1 and A_2, both guarantees are ensured simultaneously.

Example 2.24. *Consider the system in Figure 2.3. Assume we have the following contracts that specify the system in different scenarios:*

$$\mathcal{C}_1^{ag} = (A_1, G_1) = (0 \leq x \leq 5, y = 2x),$$
$$\mathcal{C}_2^{ag} = (A_2, G_2) = (x \geq 5, y = 4x).$$

Their conjunction is computed as:

$$\mathcal{C}_{12}^{ag} = \mathcal{C}_1^{ag} \wedge \mathcal{C}_2^{ag} = (A_{12}, G_{12}),$$
$$A_{12} = A_1 \cup A_2 = ((0 \leq x \leq 5) \vee (x \geq 5)) = (x \geq 0),$$
$$G_{12} = G_1 \cap G_2 = ((y = 2x \vee (x > 5 \vee x < 0)) \wedge (y = 4x \vee x < 5)).$$

The contract assumptions are combined using their union, since the system is expected to operate as long as at least one condition is satisfied. The guarantees act as selectors, enforcing the corresponding guarantee from $\mathcal{C}_1^{ag}$ or $\mathcal{C}_2^{ag}$ based on which assumption holds. For example, when $x \geq 5$, the system must satisfy $y = 4x$ to fulfill the guarantee.

Implication is the inverse operation of conjunction. Given an overall contract $\mathcal{C}_{12}^{ag}$ and a contract $\mathcal{C}_1^{ag}$ representing the specification in a scenario, the operation $\mathcal{C}_1^{ag} \rightarrow \mathcal{C}_{12}^{ag}$ computes the contract that captures the remaining scenario.

2.3.3.3 Merging and Separation

Merging is introduced to capture multiple viewpoints of a system, such as functionality, power, and timing, within a single specification. It produces a unified contract that encapsulates all these aspects. The key distinction between merging and conjunction is that merging requires that all assumptions and guarantees be satisfied simultaneously for the design to function correctly, whereas conjunction allows different guarantees to apply under different scenarios.

Definition 2.22. *The merging of two saturated contracts $\mathcal{C}_1^{ag} = (A_1, G_1)$ and $\mathcal{C}_2^{ag} = (A_2, G_2)$, denoted by $\mathcal{C}_1^{ag} \cdot \mathcal{C}_2^{ag}$, is computed as:*

$$(A_1 \cap A_2, (G_1 \cap G_2) \cup \overline{(A_1 \cap A_2)}).$$

Figure 2.5: An example system used to illustrate contract merging.

Example 2.25. *Consider the system in Figure 2.5, specified by the following contracts, representing different viewpoints:*

$$\mathcal{C}_1^{ag} = (A_1, G_1) = (T \geq 0, x \leq 2),$$
$$\mathcal{C}_2^{ag} = (A_2, G_2) = (7 \leq V \leq 10, y = 5).$$

Contract $\mathcal{C}_1^{ag}$ specifies that the temperature T must be at least 0 for the component to function properly, and it can generate a value such that $x \leq 2$. Contract $\mathcal{C}_2^{ag}$ requires the operating voltage to be between 7 and 10, and the system outputs a constant value of $y = 5$.

The merging of the two viewpoint contracts is computed as:

$$\mathcal{C}_{12}^{ag} = \mathcal{C}_1^{ag} \cdot \mathcal{C}_2^{ag} = (A_{12}, G_{12}),$$
$$A_{12} = ((T \geq 0) \wedge (7 \leq V \leq 10)),$$
$$G_{12} = ((x \leq 2 \vee T < 0) \wedge (y = 5 \vee V > 10 \vee V < 7) \vee (T < 0) \vee (V > 10 \vee V < 7)).$$

Observing the resulting contract, the assumption requires both viewpoint assumptions to be satisfied, and the guarantees enforce the guarantees from both viewpoints. The expression $\overline{(A_1 \cap A_2)} = (T < 0) \vee (V > 10 \vee V < 7)$ is included for saturation purposes and does not impose any additional obligation.

Separation is the inverse operation of merging. Given an overall contract $\mathcal{C}_{12}^{ag}$ and a viewpoint contract $\mathcal{C}_1^{ag}$, the operation $\mathcal{C}_{12}^{ag} \div \mathcal{C}_1^{ag}$ derives the contract for the missing viewpoint.

2.3.3.4 Disjunction and Coimplication

Disjunction is defined as an operation for computing the shared abstraction of multiple contracts:

Definition 2.23. *The disjunction of two saturated contracts $\mathcal{C}_1^{ag} = (A_1, G_1)$ and $\mathcal{C}_2^{ag} = (A_2, G_2)$, denoted by $\mathcal{C}_1^{ag} \vee \mathcal{C}_2^{ag}$, is computed as:*

$$(A_1 \cap A_2, G_2 \cup G_1).$$

Disjunction abstracts a family of products characterized by contracts and generates a specification for the entire product family [77].

Coimplication is the inverse operation of disjunction. Given a contract $\mathcal{C}_{12}^{ag}$ representing a product family and an existing contract $\mathcal{C}_1^{ag}$ within it, the operation $\mathcal{C}_1^{ag} \not\twoheadrightarrow \mathcal{C}_{12}^{ag}$ derives the contract for a product that is part of the family but not covered by $\mathcal{C}_1^{ag}$.

2.4 Contracts Background

This section introduces the evolution of contract theory and its adoption as a design methodology for cyber-physical systems (CPS).

Origins: A Software Engineering Perspective In 1992, Meyer [109] introduced the term *contract* in software engineering, drawing an analogy to business contracts between a function's caller and its implementation, within the context of the Eiffel programming language [110, 111]. The concept builds on the notion of preconditions and postconditions, originating from the Hoare logic [49, 56, 67], to define the requirements of a program method. In this methodology, preconditions and postconditions separate the responsibilities of the function caller and its implementation. The caller must ensure that the preconditions are satisfied before invoking the function, while the function's implementation must guarantee that the postconditions hold upon completion. These principles extend to object-oriented programming, where subclass redefinitions must not strengthen preconditions or weaken postconditions to maintain compatibility with the parent class.

Specifications for Concurrent Systems Parallel to the development of contracts in software engineering, researchers were exploring how to specify and verify concurrent systems, where each component can execute independently without waiting for others.

Inspired by Hoare logic, rely-guarantee reasoning [83] extends the concept by introducing *rely* and *guarantee* conditions for concurrent programs. Rely conditions describe the assumptions a program makes about changes to the global state by other processes, while guarantee conditions specify the changes the program is allowed to make to the global state.

Another approach uses temporal logic [137] to express specifications for concurrent systems. Pnueli [136] introduced assume-guarantee reasoning, extending preconditions and postconditions of Hoare logic into the temporal domain [138], building on Lamport's observations [92]. Abadi, Lamport, and Wolper *et al.* [1, 2, 3] formalized system specifications for transition systems as pairs of assumptions and guarantees, and proposed composition and decomposition principles for reasoning about open systems. Assume-guarantee reasoning has since become a widely used technique for hierarchically decomposing a system into verifiable subsystems, allowing system-wide properties to be proven using subsystem-level specifications and inference rules. This separation of assumptions and guarantees also supports modular model checking [38, 62], helping to mitigate scalability challenges in monolithic verification, as shown by Cobleigh *et al.* [39]. Building on this foundation, Dill [50]

introduced asynchronous trace structures, distinguishing success and failure traces and introducing the notion of *refinement*, which corresponds to conformance rather than the notion of refinement defined in the preliminaries. Wolf [177] extended this trace-based framework to synchronous systems. Negulescu [117] later proposed *process spaces*, defined as pairs of sets over executions, which closely resemble assume-guarantee contracts. This framework laid the groundwork for modern contract-based design theories and has been applied across domains such as electrical networks, control systems, and dynamic systems.

In parallel, another significant line of work focuses on *I/O automata* [101], a model of computation for asynchronous distributed networks. I/O automata describe behaviors using states and actions (inputs, outputs, and internal), along with a transition relation, different from the trace-based approaches. They are *input-enabled*, meaning that for every input action and state, a corresponding transition must exist. To support environment assumptions, De Alfaro and Henzinger [6] proposed *interface automata*, which eliminate the input-enabled requirement. This allows certain input-state combinations to represent behaviors that the environment is assumed not to exhibit. In their follow-up work [46], they further distinguished between *components*, which accept all environments, and *interfaces*, which constrain the environment. This distinction led to extensive work on interface theories, which extend interface automata to various domains, including timed interfaces [47], resource interfaces [29], permissive interfaces [65], modal I/O automata [95], interfaces for component reuse [51], timed-automata [45], and modal interfaces [143, 144]. Larsen *et al.* [94] separated implicit assumptions in interface automata, introducing *interface input/output automata*, which comprise two I/O automata: one capturing assumptions and the other specifying guarantees.

The development of these specification frameworks led to model-driven engineering [85, 97, 154]. In this methodology, a design specification is an integral part of the system architecture, and comprises typed ports, parameters, and attributes. Specifications are typically expressed as constraints on components, often using the Object Constraint Language (OCL)[174], which defines the context for each statement and specifies properties that must hold within that context.

Adoption in CPS Design Researchers in cyber-physical system design adopted these concepts from software engineering and concurrent system specifications, focusing on specifications for reactive system interfaces. Reactive systems, as defined by Harel and Pnueli [64], continuously react to inputs from their environment. Techniques for formal specification in this context often rely on the trace semantics of system behaviors.

Damm *et al.* [42] introduced the concept of the *rich component* in CPS engineering. A rich component integrates both functional and non-functional aspects in the context of model-based design. This idea laid the foundation for the application of formal methods to CPS design, leading to the development of contract-based design for CPS. Building on this work, Benveniste *et al.* [20] proposed the concept of *heterogeneous rich components*, offering the first formal definition of assume-guarantee contracts. In this context, they use the term

promise instead of *guarantee*.

Since the introduction of assume-guarantee contracts, contracts have gained significant attention from CPS researchers for their potential to enhance design methodologies. Specifically, Sangiovanni-Vincentelli *et al.* [149, 151] advocate for using contract-based design to manage system design complexity, including virtual integration for early fault detection within the V-model of model-based design. This approach is combined with platform-based design [87], where the design process progresses through multiple abstraction layers that separate functionality from architecture [122]. Nuzzo [119] envisions that design automation and contract-based design can address CPS design challenges, mirroring the success of EDA in VLSI design.

Development of Theories The commonality among these specification approaches lies in their use of a paired structure to describe assumptions about the environment and the responsibilities of the system. This principle has inspired extensive research in contract theory, a field concerned with the properties of specifications and and the enhancement of their expressiveness across diverse applications. Research in this area has focused on formalizing refinement, comparing specification frameworks, and developing operators to support system-level reasoning.

Back and von Wright *et al.* [10] introduced contracts in the refinement calculus [11], where refinement ensures the preservation of all total correctness properties. In this framework, processes are described using guarded commands that operate on shared variables. Contracts in this setting consist of assertions (higher-order state predicates) and state transformers. Unlike assume-guarantee contracts, however, this formulation does not explicitly distinguish between assumptions and guarantees, as the roles of state predicates and state transformers are not clearly defined. In contrast, Alur *et al.* [7] proposed a formal notion of refinement based on alternating simulation, which adopts a game perspective of multi-agent systems modeled as alternating transition systems. Here, the environment is viewed adversarially, requiring a component to satisfy its specification regardless of how the environment behaves. Doyen *et al.* [51] further contributed to this area by introducing the concept of shared refinement within interface theory.

As the variety of specification representations expanded, research began to focus on comparing these specifications, revealing that contract theories and interface theories could be integrated into a unified framework. Bauer *et al.* [15] demonstrated that any *specification theory* (the formalisms for properties) can form *contract theory*, regardless of the formalism of those properties. This includes examples such as the assume-guarantee contract derived from a pair of properties and modal contracts from a pair of modal specifications. Nuzzo *et al.* [123] compared interface and contract theories, proposing a transformation from interfaces to LTL assume-guarantee contracts that preserves the refinement relation, using a new assumption-projection operator to maintain the semantics of interface composition. Building on these comparisons, Benveniste *et al.* [17] developed a meta-theory of contracts that encompasses contracts, interface theory, and rely-guarantee contracts, based on a series of

research reports [16, 18, 19].

In addition to integrating these specification representations, new operators for system reasoning have been proposed to facilitate the design process. These include quotient [81, 146], merging, and separation [132]. Inigo [77] further summarized these operators and formulated their algebraic properties within contract theory.

Extension for Expressiveness In addition to theoretical advancements that improve system reasoning, various extensions of contracts have been proposed to address diverse system modeling needs. Goessler and Raclet [60], as well as Quinton and Graf [142], defined *modal contracts*, which use modal specifications to label transitions as *must* or *may*, indicating whether a transition is required or optional to satisfy the specification. Inigo *et al.* [78] introduced hyper-contracts, which express specifications over hyper-properties rather than trace properties. Oh *et al.* [130] developed *optimizing assume-guarantee contracts*, which incorporate optimization criteria to support cooperative behavior during component composition. Nuzzo *et al.* [127] presented *stochastic assume-guarantee contracts*, which consider the probability of satisfying properties in contracts. Bartocci *et al.* [14] proposed *information flow contracts* which capture system-level information flow requirements in contract form. Sievers *et al.* [161] introduced *flexible contracts*, which address unknown or unpredictable conditions at design time by employing hidden Markov models to monitor system resiliency at runtime.

Applications of Contracts in CPS Contract-based design has been applied across various CPS domains, including hybrid systems [21, 22, 116], autonomous system testing [61], space system design exploration [148], production line propotyping [164], analog circuit design [126], smart buildings [102], virtual integration of controllers [43], aircraft electric power systems [121], and control protocols [125].

2.5 Conclusion

This chapter introduced the foundational concepts essential to understanding contract-based design, It covered system modeling, formal definitions of contracts, and assume-guarantee contracts, laying the foundation for applying formal specifications in the design process. The historical overview positiond these ideas within the broader evolution of contract-based design, from software engineering to CPS. With these foundations established, the following chapters will present the theory, algorithms, and tools that enable design automation for contract-based design.

Chapter 3

Design Automation Opportunities for Contract-based Design

While assume-guarantee contracts are compact and can be constructed from any set of behaviors, they are difficult for human designers to interpret and manipulate manually due to the need for saturation and the complexity of their operation results. For instance, the composition steps and result in Example 2.22 are intricate, requiring careful examination to ensure they accurately capture the composition of the subsystem contracts. This complexity highlights the importance of design automation in easing the designers' burden and supporting the contract-based design process.

To this end, this chapter explores opportunities for automating the contract-based design methodology. First, an overview of key design automation tasks and their current research status is provided, highlighting their potential to alleviate designers' burdens, prevent design faults, and facilitate the design process. Then, the existing algorithms and tools corresponding to these design automation tasks are reviewed, followed by a summary of promising research directions for further advancements.

3.1 Challenges of Applying Contract-based Design

Contract-based design and assume-guarantee contracts, as introduced in the previous chapter, , are promising in addressing CPS design challenges by decomposing complex design problems and using contract manipulation for system reasoning. To apply the design methodology and ensure correctness throughout the design process, the following critical questions must be addressed:

Q1. **Contract Formulation**: How can contracts be effectively specified for all components within the system?

Q2. **Design Consistency**: How can the correctness of contracts be ensured, both in terms of their semantics and alignment with the design intent?

Q3. **Decomposition Validity**: What defines a desirable decomposition such that the implementations of subsystem contracts guarantee that their integrated result meets the design goal, and how can this be verified?

Q4. **Decomposition Strategy**: How can a contract be decomposed, and how can an optimal contract decomposition be achieved that leads to optimal implementations with respect to the given design objectives?

Properly addressing these questions is crucial to ensure the effective application of the methodology to optimize designs without introducing potential design faults. Additionally, efficiency is essential to prevent the methodology's benefits from being undermined by overheads it introduces, particularly those related to addressing decomposition validity and decomposition strategy questions. Consequently, design automation, with its ability to streamline the design process, is a promising framework for addressing these questions. The remainder of this chapter focuses on design automation approaches that provide answers to these questions.

Beyond the need to address these questions, designers also face several fundamental challenges in contract-based design arising from the complexity of contract manipulation, which further underscore the importance of design automation tools to alleviate these burdens. First, saturation, though essential in every contract operation, is not straightforward from the perspective of system guarantees and may be overlooked, leading to incorrect results. Automating this process would help avoid such mistakes and allow designers to focus on specifying key properties rather than performing routine tasks. Second, the similarity among set operation formulas increases the risk of errors. Automation tools can help minimize design faults by ensuring the correct application of these formulas. Lastly, the results of contract operations are often difficult to interpret. Tools that simplify or abstract these results would significantly enhance the design process.

Consequently, developing design automation tools and algorithms to address the above questions and overcome fundamental challenges is crucial for enabling contract-based design to tackle CPS design problems effectively and efficiently. The contributions of this chapter are summarized as follows:

- The essential automation tasks for enabling contract-based design are identified and categorized into specification, verification, simulation, and synthesis, each with a generalized problem formulation. These tasks are then detailed, highlighting their connections to contract theory, design methodologies, and the current research status.

- Contracts are treated as integral to the design process, with the top-level specification serving as the ultimate design goal. As the design progresses, additional information is incrementally incorporated, and automation ensures that the result of each step adheres to the top-level specification.

- A review of existing contract-based design automation tools is provided, with a comparison of their functionalities with respect to the identified automation tasks.

- Opportunities for developing new tools are identified to enhance the application of contract-based design and foster research in contract theory.

3.2 Overview of Design Automation Opportunities

In the contract-based design process, each step aims to decomposed a given contract. This decomposition narrows the design space and offers partial insight into potential implementation structures, serving as an initial step toward realizing the system.

Importantly, the distinction between specification and implementation is not absolute. Abadi [3] argues that the two are not fundamentally different: " *Formally, a specification is a set of sequences of states, which represents the set of allowed behaviors of a system. We do not distinguish between specifications and programs; a Pascal program and a temporal logic specification are both specifications, although one is at a lower level than the other.*" This layered perspective appears in many application domains. In digital circuit design, for instance, a finite-state machine (FSM) may be synthesized from a high-level specification and thus serves as its implementation. However, the FSM can, in turn, serve as a specification for a register-transfer level (RTL) design, determining the registers and the data transfers between them. The RTL design then becomes the specification for a gate-level circuit, where logic gates are composed to implement desired computations.

In design automation and platform-based design, the terms *specification*, *simulation*, *verification*, and *synthesis* are used to handle this layered design style. *specifications* define higher-level goals, while implementations are lower-level realizations intended to fulfill those goals. *synthesis* introduces additional details and constraints to transition from specifications to implementations at the next lower level. *verification* checks whether proposed implementations satisfy the requirements of their specifications, and *simulation* extracts behaviors exhibited by the current implementations. For instance, when transitioning from RTL descriptions to gate-level netlists, the RTL descriptions act as the specifications and the gate-level netlists serve as the implementations. Logic synthesis generates the netlists from the RTL descriptions. Verification ensures that the netlists conform to the RTL descriptions, while gate-level simulation reveals the behavior of the synthesized design.

Similarly, contract-based design aligns naturally with this layered approach. Contracts can serve as specifications of requirements and as characterizations of implementations, making every design artifact a contract—this is the essence of contract-based design. A higher-level contract functions as the *specification*, while its decomposition constitutes the *implementation*. Both the specification and the implementation are expressed as contracts, rather than as fundamentally distinct representations. Starting from the top-level contract, synthesis produces its decomposition, verification detects potential issues and ensures correctness, and simulation generates possible behaviors of the current implementation.

These design automation concepts enable the addressing of the critical questions in contract-based design, provided that the corresponding algorithms and tools are developed

to efficiently solve the associated problems. The following discussion presents the proposed design automation approaches for answering these questions.

3.2.1 Contract Formulation

This question focuses on contract formalisms that can express all relevant aspects of the system. The proposed solution relies on the concept of *specification*, which includes defining suitable formalisms for different types of systems and developing automation tools to support their application.

Specification is a key enabler of design automation, as it provides a well-defined problem formulation that includes the design space, design objectives, and output formats required for automation tasks. Formally representing the design problem is essential for enabling subsequent automation. To this end, appropriate formalisms must allow designers to express specifications as contracts and define the semantics of analytical operations. Contract operations support a compositional approach to specification, enabling designers to build complex specifications by combining contracts instead of writing them from scratch by combining contracts rather than writing entire specifications from scratch, thereby reducing the risk of errors.

For automation tools, it is essential to support reading, manipulating, and, most importantly, integration of different contract formalisms. Such integration provides designers with the flexibility to choose the most intuitive formalism for their domain, improving both usability and expressiveness.

3.2.2 Design Consistency

Ensuring the correctness of contracts aligns with design automation tasks in *verification* and *simulation*. For verification, when a designer specifies a contract for existing design elements, the task is to check whether the design satisfies the contract. For simulation, when a contract is written to express a design intent, the task is to verify that the translation from intent to formal specification does not introduce errors or misrepresentations. Therefore, generating and analyzing behaviors from the contract serves as a safety check, ensuring that the contract does not inadvertently include assumptions or guarantees that were not intended by the designer.

3.2.3 Decomposition Validity

One of the most attractive advantages of contract-based design is independent design. A desirable decomposition should ensure that integration results are correct, regardless of the independent design outcomes. A correct integration result means the implementation will function properly in the specified environments, and produce behaviors that satisfy the guarantees.

Contract refinement appears to offer a solution to this goal, as its transitive property guarantees that any independent design result from refinement remains a refinement of the original contract. However, refinement does not guarantee that the integration results will produce behaviors, as demonstrated by the following example:

Example 3.1. *Consider the following two contracts:*

$$\mathcal{C}_1^{ag} = (A_1, G_1) = (x \geq 0, y > 2x),$$
$$\mathcal{C}_2^{ag} = (A_2, G_2) = (x \geq -2, y > 2x \wedge x < 0).$$

$\mathcal{C}_2^{ag}$ *refines* $\mathcal{C}_1^{ag}$ *since* $A_1 \subseteq A_2$ *and* $(G_1 \cup \overline{A_1}) \supseteq (G_2 \cup \overline{A_2})$. *Any implementation* M_2 *of* $\mathcal{C}_2^{ag}$ *must be a subset of its acceptable behaviors* $M_2 \subseteq G_2 \cup \overline{A_2}$. *However,* $M_2 \cap A_1 = \emptyset$, *indicating that it produces no behaviors under the original environment* $A_1 = x \geq 0$. *This absence of behaviors suggests that the implementation fails to function properly in the required environments.*

In the above example, if contract $\mathcal{C}_2^{ag}$ results from composing decomposed contracts, it indicates that the system integrated from their implementations cannot operate together, highlighting compatibility issues. To meet design goals and avoid such problems, it is essential to ensure that all components can interact meaningfully and do not produce empty behaviors. This gives rise to a *verification* problem, which calls for formal theorems to address such issues and an automated process to carry out verification based on these theorems. Chapter 5 is dedicated to resolving the challenge illustrated by this example.

3.2.4 Decomposition Strategy

Finding a valid decomposition is fundamentally a *synthesis* problem, as it involves introducing additional information about the decomposed contracts and their interconnections. The decomposition must be derived based on the top-level specification and constrained by the available design space.

Moreover, achieving an *optimal* decomposition requires not only satisfying correctness requirements but also optimizing for specific design objectives. This requires defining evaluation criteria to compare decomposition outcomes, such as functional performance, number of subsystems, implementation cost, or other relevant design factors. Consequently, decomposition synthesis is a particularly challenging problem, as it demands the full range of design automation capabilities to identify and produce an optimal solution.

The following sections elaborate on these aspects of design automation and their corresponding problem definitions.

3.3 Contract Specification

Contract specification focuses on developing formalisms for various systems and automation tools that support their use, including manipulation and integration. Key opportunities

include enabling support for physical components and providing comprehensive support for contract manipulation.

3.3.1 Physical Component Specification

Two critical properties of physical components are the use of *implicit functions* and *flexible port directions*. Implicit functions [176] are essential because they describe physical behavior through equations that do not require explicit input-output relationships, which are often not analytical solvable. Flexible port directions support component reuse. For example, a resistor can operate in different roles depending on the scenario: it can take current as input and produce a voltage drop, or, when placed across a battery, take voltage as input and produce current. The governing implicit function is Ohm's law, which relates the port values without inferring the direction of data flow.

Therefore, specifying physical components in existing contract formalisms requires careful examination of their ability to accommodate implicit functions and ensure correctness. These formalisms, often rooted in programming language concepts, typically do not incorporate implicit functions for specification. The flexibility in port direction raises an interesting question: Can the contract for a physical design element be defined without explicitly designating inputs and outputs? If so, a physical component could be compactly represented by a single contract, reducing the need for complex expressions and allowing applicability across various usage scenarios.

Many studies have applied contracts to physical systems. Benvenuti *et al.* [21, 22] defined assume-guarantee contracts for a closed-loop water tank control system, demonstrating that contracts can be applied to hybrid systems and control. Although physical components are involved, they are modeled as input/output systems. Composition is performed manually and relies on various reduction techniques to analyze equivalent states. However, assumptions are not integrated into the composition process. They are presumed to be compatible by default, overlooking cases where input variables must be constrained to satisfy the assumptions of other components. Nuzzo *et al.* [121] later applied an assume-guarantee contract framework to the design of aircraft electric power systems. They used a dynamic behavioral model, $\mathcal{F}(U, Y, X, \kappa) = 0$, where U is the set of input variables, Y the output variables, and X the internal (state) variables, to describe component behavior. Although this approach promotes the use of implicit functions, it still relies on explicitly partitioned input and output variable sets to define contracts. As discussed in Chapter 4, assume-guarantee contracts may be unwieldy when applied to physical systems, as they implicitly enforce a notion of port direction.

3.3.2 Comprehensive Supports for Contract Manipulations

Due to the complexity of CPS, it can be challenging for designers to derive a standalone contract that accurately specifies an entire system. Instead, automation tools should support modular specification, allowing designers to express different scenarios, viewpoints, or

subcomponents of the design. Comprehensive support for contract manipulation is therefore essential to enable the dynamic creation and combining of contracts. This can be achieved by implementing contract manipulations introduced in Chapter 2. To provide such support, the importance of the automation tool is to handle the burden of saturation and address the hard-to-interpret operation result.

Contract saturation is not intuitive when considering the guarantees it provides. As a result, the need for saturation increases the risk of errors when manipulating contracts. For example, consider a specification for a design that computes the sum of two positive numbers: "*Given two positive numbers, the result should be the sum of the inputs.*" To convert this description, the input ports are assigned as x and y, and the output port as z, resulting in the contract $(x > 0 \wedge y > 0, z = x + y)$. The property $x > 0 \wedge y > 0$ translates to: "*Given two positive numbers, and $z = x + y$ corresponds to "the result should be the sum of the inputs.*" Although this intuitive encoding correctly represents the requirement, it is not saturated. Therefore, the contract must be saturated before performing contract operations. This mandatory yet non-intuitive step complicates contract manipulation, highlighting the need for tools and assistance to facilitate the saturation process.

Another important capability is the handling of hard-to-interpret operation results, which arise from the nature of the operations and the need for saturation. To address this, Inigo *et al.* [80] proposed algorithms to eliminate unnecessary ports, simplifying operation results to make them more understandable. While effective, this approach may not be suitable for all operations, as it involves theorem solving, which can be computationally expensive and unnecessary in some cases. As an alternative, operations that do not require designer input should be encapsulated and automatically handled by the tools. This approach requires tools for contract management to ensure the correctness of these automated operations.

3.4 Contract Verification

Contract verification aims to detect faults that arise during the design process. These faults may result from errors in contract specification, incorrect implementation, or bugs in automation tools. Verification can be categorized into the following tasks: *Meaningless Contract Detection*, *Refinement Verification*, *Decomposition Verification*, *Implementation Verification*, The following sections elaborate on each of these tasks.

3.4.1 Meaningless Contract Detection

This type of verification focuses on determining whether a contract is meaningful based on its formalism. Verifying meaningfulness is crucial to avoid vacuous requirements and properties, such as those identified by Armoni *et al.* [8] in the context of LTL. Examples of meaningless contracts include those that are inconsistent or incompatible, as discussed in Chapter 2. To detect such errors, one can examine whether the implementations or environments of a contract result in empty sets. Cimatti *et al.* [34] introduced functionality to check for contract

incompatibilities or inconsistencies. Non-empty contract obligations are also important, as they define the behaviors a system is expected to exhibit in its target environments. Contracts with empty obligations are therefore meaningless, as any implementation based on them cannot exhibit any behavior, even when the systems are expected to function normally.

In addition to consistency and compatibility, some applications may require additional constraints to meet domain-specific requirements. For example, in controller design, output values should depend solely on input values and must not be controlled by the environment. As a result, the ports of a system can be categorized as uncontrollable or controllable, denoted by (u, c), where u represents the uncontrollable ports and c the controllable ones. This distinction leads to the concepts of ***u**-receptiveness* and ***c**-receptiveness* [151]. A property is **u**-receptive if it accepts any combination of port values set by the environment, indicating that the system has no control over those ports. Consequently, the guarantee G for a controller should be **u**-receptive. Similarly, the assumption A should be **c**-receptive to ensure that the environment does not constrain ports controlled by the system.

The following summarizes the verification problem for meaningless contract detection:

Problem 3.1. *Given any contract $\mathcal{C} = (A, G)$, check if the contract satisfy the following properties:*

- $A \neq \emptyset$ *(Compatibility),*
- $G \cup \overline{A} \neq \emptyset$ *(Consistentency),*
- $A \cap G \neq \emptyset$ *(Non-trivial obligation),*
- *and other application-specific properties.*

3.4.2 Refinement Verification

Refinement is a critical contract relation that ensures that any implementation based on a modified contract does not violate the original requirements. Cimatti *et al.* [35] proposed a property-based proof system for verifying whether a decomposition result refines the system contract. Le *et al.* [96] introduced a general paradigm for checking decomposition conditions using $n + 1$ formulas, yielding results similar to those of Cimatti *et al.* [35] when applied to trace-based systems. These verification methods do not depend on closed-form operations of assume-guarantee contracts. Instead, they are based on the definition of composition, as introduced in Definition 2.11, and derive specific formulas to verify refinement. Antonio *et al.* [76] proposed a library-based refinement checking algorithm that utilizes abstraction, leveraging a library of pre-checked refinement relations to accelerate the verification process.

The contract refinement verification problem is formally stated as follows:

Problem 3.2. *Given a system contract $\mathcal{C}_s = (A_s, G_s)$, and a set of n proposed subsystem contracts $\mathcal{C}_i = (A_i, G_i)$ for $i = 0, \ldots, n$, determine whether the refinement relation holds by checking the following conditions:*

- $A_s \subseteq \bigcap_{i=0}^{n} A_i \cup \overline{\bigcap_{i=0}^{n} G_i}$,
- $G_s \supseteq \bigcap_{i=0}^{n} G_i$.

3.4.3 Decomposition Verification

As demonstrated in Example 3.1, refinement alone does not guarantee the correctness of contract decompositions. Establishing criteria to guarantee accurate decomposition and developing methods to verify these criteria are essential for effective contract-based design.

Westman *et al.* [175] highlighted a potential issue in which a component might vacuously satisfy a contract by exhibiting empty behavior under any environment satisfying A. To address this issue, they proposed conditions for both the supplier (who implements the component) and the client (who defines the environment). However, their conditions are stringent, requiring that the obligations be fully contained within the component, i.e., $A \cap G \subseteq M$. Such restrictions limit the supplier's flexibility in selecting behaviors that optimize performance, and thus may reduce the practical usability of contracts in real-world applications.

Consequently, identifying appropriate criteria to avoid vacuous satisfaction and developing corresponding verification methods remain open challenges. Chapter 5 will address this gap by formulating the problem and proposing solutions.

3.4.4 Implementation Verification

When an implementation of a contract, such as an actual component or its model, is proposed, implementation verification checks whether it satisfies the requirements specified by the contract. Specifically, this involves verifying that the component satisfies the property $A \implies G$.

This task can be framed as a general property verification problem, depending on the model used to describe the implementation. Consequently, any model-checking tool that supports both the modeling language and the formalism of $A \implies G$ can be used to verify the property. For instance, tools such as nuXmv [28], UCLID5 [157], Spin [69], and TLA+[93] support linear temporal logic properties. For first-order logic, commonly used tools include Z3 [48], CVC4 [13], and MathSAT [36].

The implementation verification problem can be formally stated as follows:

Problem 3.3. *Given a contract $\mathcal{C} = (A, G)$ and a component M, check whether the component satisfies the contract by verifying if the following condition holds:*

$$M \cap A \subseteq G.$$

3.5 Contract Simulation

Ensuring the correctness of the translation from design intent to contracts is a critical challenge for enabling effective contract-based design. Simulation is an essential automation task for verifying whether contracts accurately represent the designer's intent in their formal specification. The core concept of contract simulation is to generate acceptable behaviors for a given contract. Producing such behaviors provides the following benefits:

- **Contracts versus Design Intent** Simulation provides the behaviors of contract environments, implementations, and the resulting behaviors of implementations under specific environments. This capability helps designers identify deviations between the specification and their design intent. A common design fault occurs when requirements are not correctly translated into the specification. In such cases, verification alone cannot detect the issue, as it operates on an already incorrect specification. By generating simulation results for the specification, designers can assess whether it accurately represents the system's requirements. This process helps avoid lengthy design cycles caused by discovering specification errors late in the development process.

- **Lightweight Verification** Verification aims to ensure that the implementation satisfies the specification, requiring proof that all behaviors comply. However, this process is often time-consuming due to its high computational complexity. In contrast, generating a small set of behaviors typically involves less effort and can more efficiently uncover straightforward differences, which may help identify design errors. This approach has been employed in various contexts, such as in FRAIG [112], which performs circuit optimization, with one step leveraging functionally equivalent subcircuits. In this step, FRAIG uses simulation to distinguish logic gate outputs with differing functions, reducing the number of SAT solver invocations. Another example is the Scenic language [57], which uses a probabilistic language to generate scenarios for machine learning specification testing.

  For contracts, behaviors should be generated based on the allowed contract behaviors, and whether these behaviors satisfy the contract requirements should be verified.

- **Facilitating Design Correction** In addition to its benefits for verification, simulation results can provide valuable insights for correcting design errors. For example, the designer can observe unexpected behaviors, identify ports that do not behave as expected, and examine the subsystems responsible for those ports. In this way, simulation becomes an iterative process in which both tools and designers collaborate to improve the design. Even after running full verification, designers can still conduct simulations to address any failures identified during the verification process.

- **Evaluation** In addition to verification and correction, simulation enables the preliminary evaluation of system performance, especially when performance depends on specific system behaviors. One example highlighting the importance of evaluation is

its common use in VLSI design, where simulation is used to determine whether a design can run at a faster clock speed by evaluating power consumption and assessing delay impacts. Thus, evaluation can guide tools and designers in making decisions to optimize the design.

Despite the potential benefits outlined above, no existing tools or research currently support contract simulation. However, some model checking tools do offer simulation capabilities. For instance, nuXmv [28] supports the simulation of transition models using the MathSMT solver. Such tools could provide a foundation for this research, as contract simulation requires the generation of behaviors that satisfy the contracts. To reason about these behaviors, set-level reasoning is needed to accommodate contracts from different modeling languages and formalisms, along with solvers capable of reasoning about these languages and formalisms to generate satisfying port values

Chapter 6 will address this gap by formulating the contract simulation problem and proposing corresponding solutions.

3.6 Contract Synthesis

So far, none of the automation tasks mentioned above address the generation of decompositions to support the design process. Therefore, contract synthesis plays a crucial role in contract-based design by mapping design requirements from an abstract layer to a more refined one.

Generally, contract synthesis can be formally defined as follows:

Problem 3.4. *Given a contract $\mathcal{C}$ and an objective function, denoted by f, over sets of contracts, find a set of contracts $\{\mathcal{C}_1, \mathcal{C}_2, \ldots, \mathcal{C}_k\}$ such that:*

1. *The set forms a decomposition of $\mathcal{C}$.*
2. *The decomposition is optimal with respect to f.*

In short, the synthesis problem aims to decompose a system contract into a set of subsystem contracts that preserve implementation correctness and optimize a given objective function within the design space. Due to the problem's inherent complexity, additional constraints are often introduced to simplify the synthesis process. These constraints lead to different variants of the problem, including *Correcting Refinement*, *Contract Library Selection*, and *Implementation Synthesis*.

3.6.1 Correcting Refinement

In this variant, a set of subsystem contracts $\mathcal{C}_i = (A_i, G_i)$ for $i = 0, \ldots, n$ is given, and modifications are required to establish or preserve the refinement relationship with the system contract. This type of problem can be further divided into two subproblems, depending on

whether the given contracts already satisfy the refinement relation:*refinement fixing* and *refinement tightening*.

Refinement fixing focuses on establishing the refinement relationship from any given set of subsystem contracts. In this case, the composition of the subsystem contracts does not refine the system contract. The objective is to modify or augment the set so that its composition satisfies the refinement relation.

Two approaches have been proposed to address incorrect refinement. One approach leverages the quotient operator to find a missing component specification $\mathcal{C}_{n+1} = (A_{n+1}, G_{n+1})$ such that its composition with the provided subsystem contracts satisfies the refinement relation with respect to the system contract. Inigo *et al.* [81] derived a closed-form formula for the quotient of assume-guarantee contracts and later extended this into a theory for quotients [146].

The other approach involves modifying the provided contracts. Le *et al.* [96] proposed algorithms for correcting refinement by adjusting the given contracts. They introduced two strategies: the aggressive strategy and the incremental strategy. The aggressive strategy fixes refinement by enlarging the assumptions and shrinking the guarantees of each contract according to the required behaviors, ensuring refinement. This strategy is particularly useful in distributed contexts, as it does not require waiting for updates from other contracts. The incremental strategy, in contrast, iteratively updates one contract until the refinement relation is established. This approach minimizes unnecessary synthesis efforts and can reduce the number of costly set complement operations.

Refinement tightening removes redundant subsystem contracts and abstract them while preserving the refinement relationship. Given a system contract and a set of subsystem contracts whose composition refines the system contract, this process eliminates redundancy and enhances design flexibility by adjusting over-constrained assumptions and guarantees. This is particularly useful when subsystem contracts are manually created by designers without proper optimization, as fully optimizing the general synthesis problem is often computationally infeasible.

Cimatti *et al.* [32, 33] proposed an algorithm for refinement tightening, which introduces parameters into formulas through LTL operator properties. For instance, the formula $(a < b)$ can be weakened to $(a \geq b)$, leading to a parametric formula $(p_1 \implies (a < b)) \wedge (p_2 \implies (a \geq b))$, where Boolean parameters p_1 and p_2 control the degree of weakening or strengthening. The algorithm converts the problem into a multi-parameter validity synthesis task, using model-checking tools to determine the parameter values. The approach automatically removes redundant subsystem contracts and modifies them to optimize the refinement.

3.6.2 Contract Library Selection

Contract library selection exemplifies platform-based design within contract-based design. In this approach, components at a refined level of abstraction are represented by contracts, which serve as specifications for lower levels of abstraction based on their behaviors at the

current level. Given a library of contracts L, the goal is to select a subset of contracts from the library and generate port connections that satisfy the refinement relation.

Many researchers have worked on contract selection within a library. Some refer to this problem as component selection, since contracts can be used to characterize components. Peter *et al.* [133] introduced an SMT-based component synthesis approach that encodes component selection into SMT formulas to satisfy system properties. Mishra and Jagannathan [113] proposed a bi-directional, specification-guided synthesis procedure with conflict-driven learning for components specified using Hoare-style pre- and post-conditions. Iannopollo *et al.* [73] introduced a counterexample-guided inductive synthesis (CEGIS)-based flow for composing and selecting contracts from a library of components specified using LTL contracts. In their follow-up work [71, 72, 75], they refined this flow by decomposing contracts to reduce complexity. Dos Santos *et al.* [153] proposed CONDEnSe, which uses an SAT-based generator (Computation Design Synthesis, CDS) to find connections between components and a contract-based verifier to check the satisfaction of system specifications. In their subsequent work [152], they combined the generator and verifier in a CEGIS-based synthesis approach to select components that satisfy the goal. Oh *et al.* [129] presented a parameter-based synthesis method that explores contract parameters using bi-level optimization to minimize the cost function while ensuring robustness. Wang *et al.* [173] proposed a hierarchical contract-based synthesis framework that selects contracts from a well-formed library containing information on conditional refinement and composition relations between contracts. Recently, Xiao *et al.* [179] presented ContrArc, which explores CPS architecture and selects implementations by formulating connection, flow, and timing requirements, using mixed-integer linear programming with design space pruning through subgraph isomorphism.

Despite numerous contributions in this area, several limitations persist in the existing methods. For example, the works by Peter *et al.* [133], Mishra and Jagannathan [113], Dos Santos *et al.* [152, 153], and Wang *et al.* [173] focus on generating selections without addressing optimization. While the approaches by Iannopollo *et al.* [73, 75], Oh *et al.* [129], and Xiao *et al.* [179] address optimization, they restrict their objective functions to parameters associated with the contracts. Whether these restrictions are practical for general CPS problems remains an open question. Chapter 7 will explore the drawbacks of these methods and propose a new approach for library selection.

3.6.3 Implementation Synthesis

Implementation synthesis produces actual components, such as programs, controllers, and physical elements, that serve as implementations of contracts. Traditionally, engineers have designed components based on domain expertise to satisfy specifications and then verified whether the components meet the requirements. In design automation, formal synthesis focuses on the automated conversion of formal specifications into components. Examples of formal synthesis include program synthesis [107] and reactive synthesis [31, 139], depending on the application domain. Reactive synthesis, in particular, is a major focus for the CPS community [90, 91, 135, 145], as controllers are critical components of CPS. It can be modeled

as a two-player game: the environment tries to violate the specification, while the controller seeks to ensure its satisfaction. The synthesis is successful if a strategy can be found that enables the controller to satisfy the specification, regardless of the environment's actions.

For general LTL properties, the reactive synthesis problem has doubly exponential complexity. However, the problem based on a subset of LTL known as *generalized reactivity* (1) (GR(1)) has polynomial complexity in terms of the number of input and output variables [135], enabling efficient synthesis. Given a GR(1) specification, numerous solvers and digital design synthesis tools can generate a finite-state automaton that represents the system's control strategy [24, 25, 82, 140, 178].

For assume-guarantee contracts, the separation of assumptions and guarantees enables a unique synthesis approach. Chatterjee and Henzinger [30] introduced assume-guarantee synthesis, which involves the co-synthesis of two systems. This approach assumes that each system focuses on meeting its specifications, resulting in a relationship that is neither fully collaborative nor entirely competitive. Several synthesis techniques have been developed to leverage assume-guarantee reactive synthesis and enhance the synthesis process [54, 59, 86, 103, 134, 165].

3.7 Tools for Contract-based Design Automation

Automation tools encapsulate algorithms for various tasks, integrating them into a unified interface for ease of use. Given the opportunities for automation, tools for contract-based design automation are crucial for supporting these tasks and providing a seamless interface across different stages of the design process. A key factor in their effectiveness is extensive support for various tasks, which eliminates the overhead of format conversion and interfacing between tools. Consequently, developing theories, algorithms, and tools is critical to improving both the efficiency and correctness of the contract-based design process. This section reviews existing contract-based design automation tools, compares their functionalities, and identifies gaps where many opportunities remain unaddressed or lack integrated solutions.

3.7.1 Existing Tools

Early contract-based design tools primarily focused on system-level verification tasks, lacking support for broader contract operations or additional functionalities. Cofer *et al.* [40] developed AGREE (Assume Guarantee Reasoning Environment), the first contract-based design tool, under DARPA's META research program. AGREE supports AADL [53] and SysML [131] modeling languages for formally specifying system designs. It supports verification of linear-temporal logic properties using a circular compositional reasoning framework that applies induction over time. Cimatti *et al.* [34] introduced OCRA, a tool for verifying temporal contract refinements, leveraging their SMT-based proof obligations [35]. Built on NuSMV3 as a temporal logic solver, OCRA uses the Othello System Specification (OSS) format to describe components' interfaces (ports and parameters) and the desired refinement

relations for their decomposition. The OSS format incorporates their custom Othello property specification language for expressing hybrid trace behaviors in contract assumptions and guarantees. Both AGREE and OCRA focus on verifying contract refinement relations from a system-level perspective, ensuring that decomposed contracts satisfy the refinement criteria. However, they do not utilize the contract operations proposed by Benveniste *et al.* [20] and instead rely on a general concept of composition [17], possibly due to the relatively immature state of contract theory research at the time, which led to independently developed approaches for these functionalities.

With the maturation of contract theory and operations, the refinement problem has been simplified to verifying two set relations, paving the way for tools that support synthesis and contract operations. Iannopollo *et al.* [73] developed PyCo and PyColite, tools for constrained synthesis using contract libraries. PyColite, a Python package, interfaces with SMT solvers and model checkers to support contract operations such as composition and refinement. Building on PyColite, PyCo enables constrained synthesis by generating connections and selecting components based on port constraints, system requirements, and a formalism-independent objective function, ensuring that the synthesis results satisfy the system requirements. Nuzzo *et al.* [124] developed CHASE, a contract-based requirements engineering tool for cyber-physical systems. The tool features a front-end formal specification language and a back-end reasoning tool. The front-end specification describes the networks spanning different domains, connections of system components within the same domain, and contracts that specify requirements for those connections. These specifications can be automatically parsed from natural languages using English Slot Grammar [108]. The back-end reasoning tool supports contract manipulations such as composition, conjunction, refinement, and implementation synthesis, utilizing the TuLiP toolbox [178]. Santos *et al.* [153] developed CONDEnSe, a tool for contract-based synthesis that identifies component connections and verifies the correctness of architectures through contract composition and refinement. CONDEnSe takes input from a SysML-inspired domain-specific language that describes components and the system goal, then explores feasible connections between components. The selected connections define a candidate architecture configuration. The tool encodes connection constraints using SAT and generates candidate configurations, each of which is verified through contract refinement to ensure it satisfies the system goal. Mallozi *et al.* [106] developed CROME, a contract-based specification tool designed for robotic missions. CROME organizes mission goals into a library and refines mission specifications using a contract-based goal graph (CGG), which represents how a mission can be achieved through the composition and conjunction of contracts. The tool searches for a contract that refines the leaf nodes of the CGG, and then map the entire mission specification using the goal library. Each goal can then be synthesized independently through reactive synthesis. Recently, Incer *et al.* [79] developed Pacti, a contract operation tool that supports automatic port projection, enabling the elimination of irrelevant ports from operation results. This capability enhances the interpretability of contract operation results, making them more accessible to human designers. Although the tool currently supports only contracts specified using polyhedral sets, it has been successfully applied in space missions [148], demonstrating

the practical value of removing unrelated ports to improve human comprehension.

3.7.2 Comparisons of Existing Tools

This section compares the capabilities of existing tools based on the automation opportunities introduced in this chapter. Since *Implementation Verification* and *Implementation Synthesis* are application-dependent tasks involving implementations, they are excluded from the comparison. Instead, the focus is on automation tasks where both inputs and outputs are represented by contracts. Table 3.1 compares the tools' support for system automation tasks, while Table 3.2 provides details on their support for contract manipulations.

		OCRA	CONDEnSe	PyCo	AGREE	CHASE	CHROME	Pacti
Spec.	I/O Systems	O	O	O	O	O	O	O
	Physical Systems	X	X	X	X	X	X	X
	Contract Manipulations	See Table 3.2						
Veri.	Meaningless Contract Detection	O	X	O	X	X	X	X
	Refinement Verification	O	O	O	O	O	O	O
	Decomposition Verification	X	X	X	X	X	X	X
Sim.	Contract Simulation	X	X	X	X	X	X	X
Syn.	Correcting Refinement	X	X	X	X	X	X	O
	Contract Library Selection	X	O	O	X	X	X	X

Table 3.1: Comparison of automation task support across existing contract-based design automation tools.

		OCRA	CONDEnSe	PyCo	AGREE	CHASE	CHROME	Pacti
Properties	Consistency	O	X	O	X	X	O	X
	Compatibility	O	X	O	X	X	O	X
Operations	Composition	X	O	O	X	O	O	O
	Quotient	X	X	X	X	X	X	O
	Conjunction	X	X	X	X	O	O	X
	Implication	X	X	X	X	X	X	X
	Merging	X	X	X	X	X	X	O
	Separation	X	X	X	X	X	X	X
	Disjunction	X	X	X	X	X	X	X
	Coimplication	X	X	X	X	X	X	X
Relations	Refinement	O	O	O	O	O	O	O
	Conformance	X	X	X	X	X	X	X
	Strong Dominance	X	X	X	X	X	X	X

Table 3.2: Comparison of contract manipulations support across existing contract-based design automation tools.

Existing tools cover only a small subset of automation tasks. Most are limited to contracts for I/O systems, supporting tasks such as refinement verification and contract library

selection. These contracts rely on fixed port directions, lacking flexibility, which makes them unsuitable for physical systems. Outside of these tasks, Pacti [79] is the only tool capable of correcting refinement using the quotient operator. OCRA [34] and PyCo [73] can detect consistency and compatibility. However, no tools currently support *Physical Component Specification* with implicit functions and flexible port directions. Additionally, there is a lack of support for functionalities such as *Decomposition Verification* and *Contract Simulation*.

A similar lack of support exists for contract manipulations. Most tools focus exclusively on contract composition, refinement, consistency, and compatibility. Some recent tools offer limited additional operations: Pacti [80] supports quotient and merging, while CHASE [124] and CROME [106] support conjunction. For Pacti, the focus on variable elimination, which is crucial for composition and quotient operations where intermediate ports are removed, explains the tool's prioritization of supporting composition and quotient. As a result, no tool currently offers a comprehensive set of contract manipulations despite the availability of closed-form formulas in the literature [77, 81, 132].

3.7.3 Opportunities for Contract-based design Automation

In light of the insufficient functionalities in contract manipulations and automation tasks, new theories, algorithms, and tools are necessary to bridge the gap and facilitate both research and application in contract-based design. This section details these opportunities for advancing contract-based design automation.

3.7.3.1 Theories

Contract theories have been extensively developed, offering a rich set of operations, relations, and properties to support system reasoning. However, as discussed in the aforementioned automation tasks, important gaps remain, particularly in areas such as physical system specification and decomposition verification. Addressing these gaps is essential for streamlining the design process and ensuring the correctness of implementations.

First, because assume-guarantee contracts implicitly enforce fixed port directions, they are unwieldy when applied to physical systems. To address this limitation, a new contract formalism is needed to support ports with flexible directions. Once such a formalism is established, corresponding theories for its manipulation, methods for comparing it with assume-guarantee contracts, and conversion techniques will be necessary to designers to work seamlessly across both formalisms.

For decomposition verification, it is necessary to investigate the conditions leading to the scenario in Example 3.1. Based on these conditions, theories will need to be developed to constrain refinement and avoid incorrect decomposition. These constraints will then be integrated into the contract-based design methodology, alongside the refinement relation.

3.7.3.2 Algorithms

Algorithms are essential for automating contract-based design tasks. While theoretical advances help close foundational gaps, practical implementation requires effective algorithms. Currently, there are no algorithms that support contract simulation, and, as discussed in Section 3.6, existing contract library selection methods suffer from various limitations. Developing new algorithms to overcome these limitations could broaden the scope of synthesis applications. Therefore, key opportunities lie in developing algorithms for physical component specification, decomposition verification, simulation, and more general contract library selection.

A crucial goal for these algorithms is general implementability and abstraction, rather than being narrowly designed for specific applications. Since contract-based design is an abstract concept, the algorithms must be expressed using set operations to ensure applicability across different background theories. While specific algorithms may exist for certain background theories, offering better efficiency, they may not be universally applicable. Such specialized algorithms should be regarded as optimizations for particular contexts, rather than the foundation of general contract-based design algorithms.

3.7.3.3 Tools

Finally, the developed algorithms should be integrated into tools with well-defined interfaces to ensure ease of use for designers. These tools should support contract manipulation, enable conversion between different formalisms, and facilitate the automation tasks discussed earlier to improve efficiency and ensure correctness. Most importantly, they should encapsulate contract operations to manage specifications effectively and prevent the display of hard-to-interpret results that may confuse users. Such features are essential for building tools that are both broadly applicable and accessible, especially for designers who are not experts in contract-based design.

3.8 Conclusion

This chapter discussed the challenges of applying contract-based design and explored the opportunities for automation tasks to address these challenges. The automation tasks are categorized into specification, verification, simulation, and synthesis based on their characteristics. A review of existing algorithms and tools reveals significant gaps in facilitating contract-based design and ensuring correctness. As a result, new theories, algorithms, and tools must be developed to address these gaps, presenting substantial research opportunities.

Chapter 4

Specification: Contract Formalisms for Physical Systems

This chapter introduces *constraint-behavior contracts*, a formalism designed for specifying physical components with implicit functions and flexible port directions. The operations and relations between constraint-behavior contracts are defined to facilitate system reasoning without port directions. The capability of constraint-behavior contracts to integrate with assume-guarantee contracts gives the user the choice of a formalism to use at different abstraction layers. A case study based on an Unmanned Aerial Vehicle design problem shows that the proposed constraint-behavior contracts can facilitate system verification by expressing physical components, reducing the number of contracts, and providing an intuitive encoding of contracts.

4.1 Introduction

Cyber-Physical Systems (CPS) are an integration of both physical parts and cyber parts [99]. A physical part is one realized in matter, with shapes, and mass, and can be observed directly in physical quantities in real life. In contrast, a cyber part is a logical operation or communication mechanism such as software and algorithms. As the need for large-scale CPS increases in applications such as autonomous vehicles, Industry 4.0, and smart grids, the complex interaction between the heterogeneous parts in CPS causes the prolonged and error-prone design process and thus result in prohibitively high costs.

Contract-based design (CBD) is a system development methodology that relies on contracts—formal specifications that define the expected environments and implementations—to enable correct-by-construction design, incorporate different design aspects, and reduce design complexity [119, 149, 151, 158]. The methodology leverages abstraction and refinement of the specifications to address complex design challenges. Abstraction simplifies the specification by relaxing non-critical details, making the design process more manageable. Refinement, on the other hand, reintroduces constraints for the previously relaxed aspects, bringing the

specification closer to implementation. These two processes can be performed hierarchically without violating the original top-level specification: subsystems that refine a system can be further refined individually, guaranteeing satisfaction of the system specification. The design of each subsystem thus becomes an independent design problem, allowing the designers to focus on smaller tasks and reuse previous designs when the same one arises. As a result, the methodology facilitates efficient design exploration at various levels of detail while keeping the problem size manageable. Furthermore, different design aspects (viewpoints) can be defined separately and then easily combined through contract operations [20, 132], further reducing the complexity introduced by heterogeneous design aspects.

With the potential to address challenges in CPS design, the formalism of contracts and contract-based design for CPS has attracted significant research interest [43, 121, 126, 164]. Among many formalisms, assume-guarantee contracts stand out as particularly promising candidates due to their compactness and ease of use. An assume-guarantee contract is a pair of assumption and guarantee sets $\mathcal{C} = (A, G)$ [20]. Its semantics state that when the environment behavior satisfies the assumption set A, the specified component produces behaviors within the guarantee set G. The acceptable behaviors allowed by the contracts are, therefore, $G \cup \overline{A}$, and the behaviors under the environments E are $E \cap (G \cup \overline{A})$. Various operations of assume-guarantee contracts have been proposed to reason about the system when composing components, incorporating viewpoints, and considering different operating conditions [20, 77, 81, 132].

To leverage contract-based design in the CPS design process, every component involved must be associated with a contract, including both cyber and physical components. The cyber components, such as networks and control algorithms, monitor the system state and control the physical components accordingly. Physical components, like sensors and actuators, interact with the environment and create feedback loops with cyber components to perform specific actions. Therefore, the ability to express specifications for these diverse components and their interactions in contracts is crucial for fully realizing the benefits of contract-based design.

Physical components are typically modeled through implicit equations [176], while cyber components are commonly defined by an explicit input-output relationship that establishes a data flow. As shown in Figure 4.1, electronic and mechanical systems are usually described by a system of implicit equations with more than thousands of variables. Examples of popular modeling tools include Simulink [168], which utilizes a signal-flow-based model that requires an explicit relationship between input and output, and Modelica [58], which uses equation-based models that do not need an explicit relation. Converting an implicit equation into an explicit one requires either solving the equations in closed-form solutions or using a numerical algorithm when a closed-form solution does not exist. Figure 4.2 illustrates an example where the explicit relationship between the output voltage and the input voltage necessitates solving a differential equation. As a result, the ability to express the properties of a physical system using implicit equations is crucial to avoid the complexities of equation solving. Therefore, an ideal contract formalism for physical components should provide desired semantics and accommodate implicit relationships between the variables.

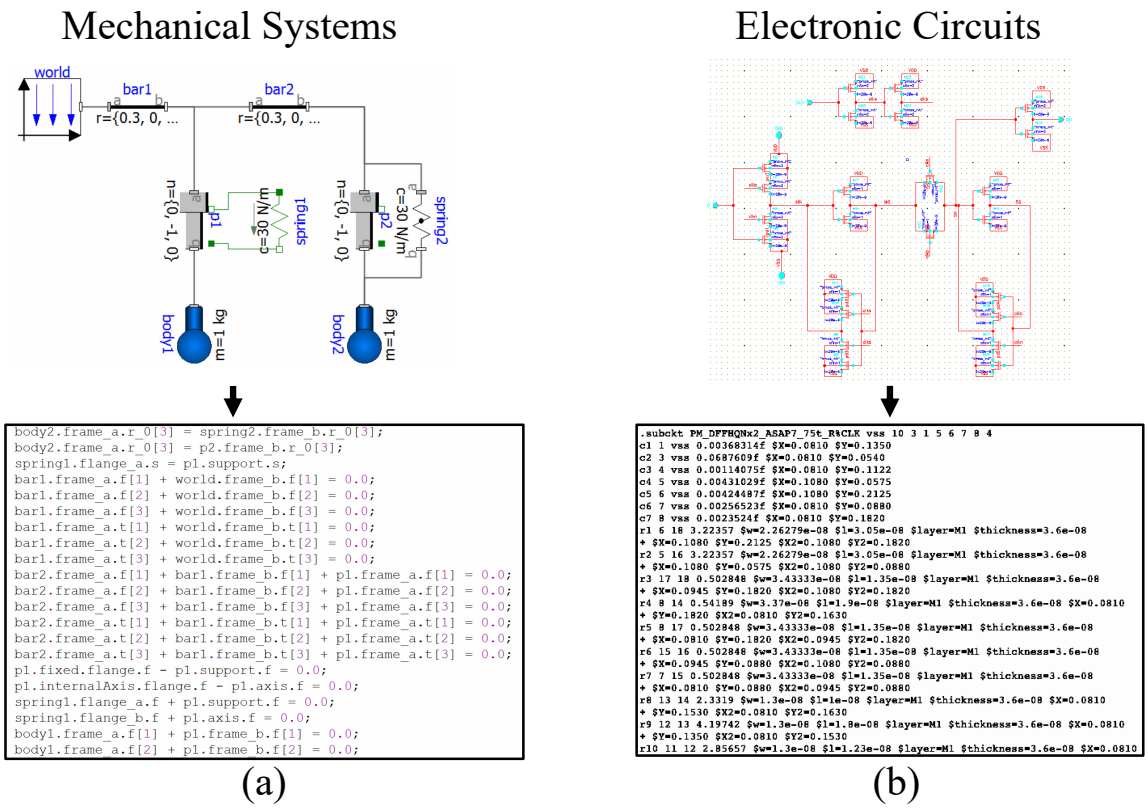


Figure 4.1: Two examples of systems that use many implicit equations for modeling: (a) a Modelica example model of a spring mass system [58], and (b) a SPICE model of a parasitic extracted D Flip-Flop from the ASAP7 Design Kit [37].

Although the theory of assume-guarantee contracts does not restrict expressions to input-output relationships and can accommodate implicit equations, the intuitive ways of specifying physical components in assume-guarantee contracts can lead to undesired semantics when the environment does not control all variables included in the assumption. As a result, the environment must control all variables in the assumption, which implicitly defines the ports corresponding to the assumption variables as the input ports of the component. We refer to this phenomenon as the *implicit port directions* issue in assume-guarantee contracts. Figure 4.3 illustrates the implicit port directions issue for a resistor—a component that allows for different port directions. The resistor, as shown in Figure 4.3 (a), can take voltage as input and current as output, and it can also take current as input and provide current as output. Therefore, the port directions of the resistor depend on the environment, indicating that its port directions are not inherently defined. One may formulate the assume-guarantee contracts for the resistor as the three formulations shown in Figure 4.3 (b). In the example, the environment E_1 does not control the current I, while the assumptions in formulations $\mathcal{C}_1$ and $\mathcal{C}_2$ contain the variable I. Consequently, these contracts allow behaviors such as

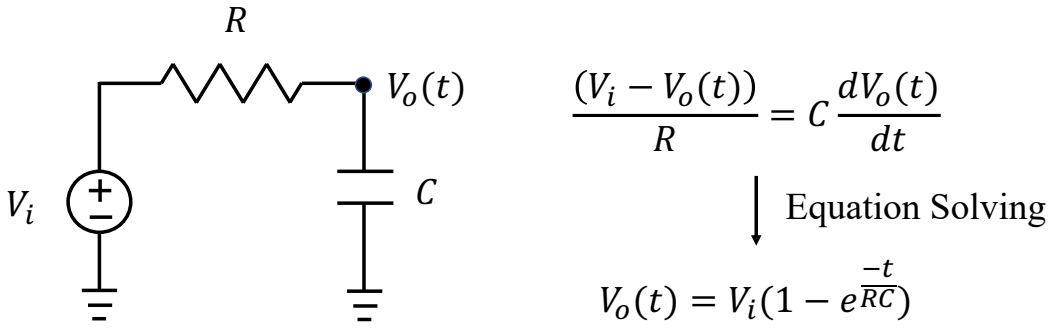


Figure 4.2: An example showing that even when the port directions of individual components are known, the composed system is expressed in terms of implicit equations and requires solving equations to convert it to an explicit expression.

$(V, I) = (2, 10)$ because the component is unconstrained when the current values violate the assumptions. However, the resistor should only produce the behavior $(V, I) = (2, 1)$ when the voltage is $2V$, indicating the semantics of the contracts are incorrect in the environments. Similarly, contracts $\mathcal{C}_1$ and $\mathcal{C}_3$ fail to represent desired behaviors when the environment E_2 does not control the voltage V. As a result, all the example contracts can only be applied to environments that match their implicit port directions and fail to provide desired semantics in various environments that control different ports.

The failure to provide desired semantics in various environments leads to several disadvantages, including increased complexity in formulation, reduced compactness of expressions, and limited applicability. First, implicit equations must be converted to explicit equations that reflect the input ports for assumptions, requiring additional effort and can result in a formulation that loses its physical meaning. For example, the assumptions in the formulations $\mathcal{C}_2$ and $\mathcal{C}_3$ in Figure 4.3 cannot be easily observed as a power constraint. Second, the size of the contract library and the complexity of using contracts increase, as a contract cannot be reused for environments with different inputs. Designers must create multiple contracts with various combinations of port directions and select the one that matches the inputs from the environments. Finally, contracts cannot be utilized when the port directions cannot be defined, as this prevents the selection of appropriate contracts without knowledge of the environment.

This chapter addresses these issues by proposing a new compact contract formalism called constraint-behavior contracts. To the best of our knowledge, this is the *first* approach that explicitly considers contract formulations for physical components governed by implicit equations. The formalism has the following properties:

- Constraint-behavior contracts are invariant to port directions, which allows the contract library to be compactly created without solving implicit equations and enumerating combinations of port directions.

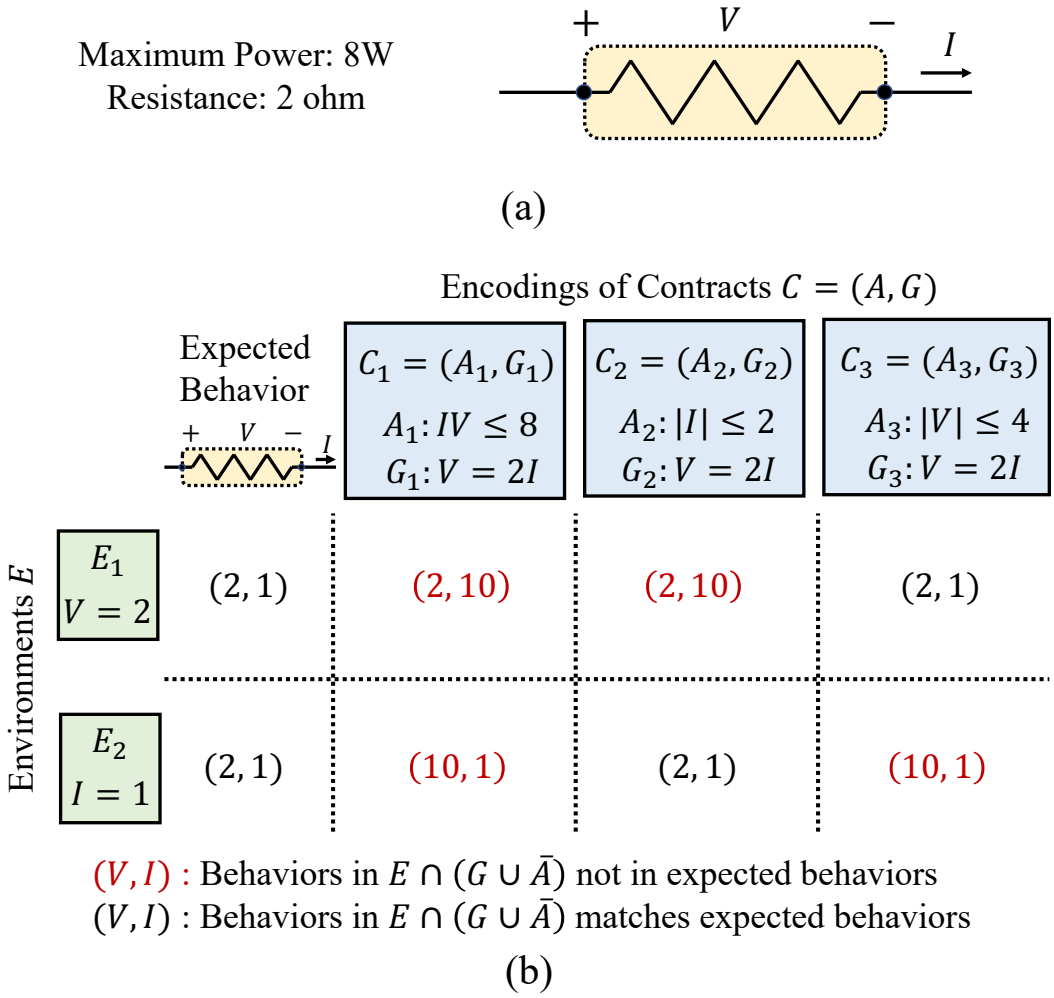


Figure 4.3: Implicit port directions in assume-guarantee contracts: (a) a resistor with maximum power constraint as a motivating example component and (b) three assume-guarantee contract formulations for the resistor. The contract expresses the behaviors (V, I) correctly only when the actual input ports match the ports for defining the assumption.

- The physical meaning of system requirements is preserved by the implicit equations in constraint-behavior contracts. Preserving physical meaning in contracts allows designers to formulate contracts and discover potential design faults more easily.
- Constraint-behavior contracts can be integrated with assume-guarantee contracts in the contract-based design process. The approach is exemplified by a demonstration based on an Unmanned Aerial Vehicle (UAV) system design verification problem.

The remainder of this chapter is organized as follows. Section 4.2 presents the Constraint-behavior contracts. In Section 4.3 , we discuss the properties of the proposed contracts under certain axioms that enable meaningful and consistent operations with assume-guarantee contracts. In Section 4.4, we further propose a special operation to combine multiple physical models that represent different operating conditions. Section 4.5 compares constraint-behavior contracts with assume-guarantee contracts. Next, in Section 4.6, we introduce the verification methodology for using constraint-behavior contracts with assume-guarantee contracts. We demonstrate the application of constraint-behavior contracts in a UAV propulsion system design verification in Section 4.7. Section 4.8 concludes this chapter.

4.2 Constraint-Behavior Contracts

This section first discusses why assume-guarantee contracts implicitly impose port directions. Then we introduce constraint-behavior contracts to address the difficulties that implicit port directions present to physical components. Operations and relations for constraint-behavior contracts are also introduced.

4.2.1 Port Sensitivity and Implicit Port Directions

In our discussion of Figure 4.3 in Section 4.1, we observed that assume-guarantee contracts may implicitly require port directions, which makes expressing behaviors in implicit equations challenging. We analyze the cause of implicit port directions in assume-guarantee contracts and then present the requirements for a contract formalism that does not imply port directions.

First, we introduce the notion of port sensitivity.

Definition 4.1. *A set of behaviors A is said to be* insensitive *to a port if the behavior of the port does not affect A. If it does then A is* sensitive *to the port.*

If a port is not used in the expression of the behavior set, the behavior set is insensitive to the port. For example, considering a system containing two resistors with resistances R_1 and R_2, respectively, the behavior set defined by Ohm's law $V_1 = I_1 R_1$ is sensitive to the voltage V_1 and current I_1. On the other hand, the behavior set defined by Ohm's law of the other resistor $V_2 = I_2 R_2$ is insensitive to the voltage V_1 and current I_1, as any values of V_1 and I_1 do not affect whether a system behavior is contained by the behavior set.

When an environment controls all the ports that the assumption set is sensitive to, the satisfaction of the assumption is determined by the behavior set of the environment on these ports. Let E denote the behavior set of the environment. If the environment satisfies the assumption, the resulting behavior must fall within the intersection of the behavior of the environment and the guarantees, denoted as $E \cap G$, since an assume-guarantee contract requires the system to ensure the specified guarantee when the assumption is satisfied. On

the other hand, if the environment violates the assumption, denoted as $E \not\subseteq A$, the system is not required to provide any guarantees.

However, when some ports that the assumption set is sensitive to are not controlled by the environment, the satisfaction of the assumption depends on the behavior of the uncontrolled ports, which can be any behavior in the universe of the port behavior. Therefore, the specified component is allowed to produce any behavior, since the assumptions can be violated by the behaviors of the uncontrolled ports. The contract thus fails to represent the components for our purpose because it always contains behaviors that the component should not produce under the environment. As a result, assume-guarantee contracts implicitly require the component to define all ports that the assumption set is sensitive to as input ports.

In the example shown in Figure 4.3, the environment E_1 does not control the current I; hence, the assumptions in formulations $\mathcal{C}_1$ and $\mathcal{C}_2$ are sensitive to I. Consequently, these contracts allow behaviors such as $(V, I) = (2, 10)$ because they do not specify the component when the current values violate the assumptions. However, the resistor should only produce the behavior $(V, I) = (2, 1)$ when the voltage is 2 V.

Thus, it is necessary to define a contract formalism that allows us to specify components regardless of which ports are controlled by their environment. The requirements of such formalism are as follows:

1. The responsibilities of the components, i.e. the required behaviors, should be considered before checking the satisfaction of the working condition. This aspect is the main difference between specifications using implicit equations and explicit equations. By considering the responsibility of the components first, we obtain the possible behavior for the ports not controlled by the environment, instead of allowing arbitrary behaviors that may violate the specified conditions.

2. The contracts should have a compact encoding and be easy to use, similar to assume-guarantee contracts. As the specification is the initial design stage, designers need to write, interpret, and examine the specifications. Ease of use and compactness will enable designers to express their intentions clearly and identify potential faults effectively.

3. The new contract formalism should seamlessly integrate into the design flow with contracts having port directions, such as assume-guarantee contracts. Depending on the application's requirements, the designer can define port directions for the subsystem created by physical components. For example, in a propulsion system, the control values can be treated as inputs, while the generated thrust can be treated as its output. Integration of these specifications allows moving between different abstraction layers. The overall system can be specified based on the port directions that reflect its usage, even if the components do not have port directions.

4.2.2 Constraint-behavior Contracts

Based on the requirements above, we propose *constraint-behavior contracts* as a formalism for specifying physical components.

We use *constraints* and *intrinsic behaviors* to describe physical components. The behaviors are the responsibility of the component, typically expressed in physical quantities. Constraints define the conditions under which the behaviors apply. For example, the resistor of Figure 4.3 imposes the behaviors $V = IR$, but it operates under the constraints $VI \leq P$.

As long as the system behavior following the intrinsic behaviors satisfies the constraints, the component functions as specified, producing the behavior according to its intrinsic behaviors. If the system behavior following the intrinsic behaviors does not satisfy the constraints, the component fails, and the resulting behavior becomes unspecified, except for the value controlled directly by the environment.

We define constraint-behavior contracts and their semantics as follows:

Definition 4.2. *Let $\mathcal{P}$ be the system ports, C be a set of behaviors called constraints, and B be a set of behaviors called intrinsic behaviors. A constraint-behavior contract is a pair of constraints and intrinsic behaviors denoted as $\mathcal{C}^{cb} = (C, B)$, where $C \subseteq \mathcal{B}_{\mathcal{P}}$ and $B \subseteq \mathcal{B}_{\mathcal{P}}$.*

The behaviors of a constraint-behavior contract under an environment $E \subseteq \mathcal{B}_{\mathcal{P}}$ are

$$ite(\emptyset \subset E \cap B \subseteq C, E \cap B, E),$$

where ite() is the IF-THEN-ELSE (ITE) operator.

The semantics of constraint-behavior contracts differs from that of assume-guarantee contracts. Observe that the responsibility of the constraint-behavior contracts is applied first to verify that the specified behaviors satisfy the constraints, which define the working condition of the components. This fulfills our first requirement for the contract formalism for physical components.

The following example shows a constraint-behavior contract for specifying the resistor in Figure 4.3 (a):

Example 4.1. *We can write the constraint-behavior contract for the resistor as $\mathcal{C}_r^{cb} = (C_r, B_r)$:*

$$C_r : IV \leq 8 \text{ and } B_r : V = 2I$$

We can verify that its semantics align with our intuition of a resistor. When the environment provides $V = 2$, we apply the intrinsic behaviors to the environment's behaviors and observe that the constraints are satisfied: $E \cap B_r = (V, I) = (2, 1)$, which is a proper subset of $C_r = (IV \leq 8)$. Thus, the behaviors of the constraint-behavior contract under this environment are $E \cap B_r = (V, I) = (2, 1)$. Similarly, when the environment provides $I = 1$, we apply the intrinsic behaviors to the environment's behaviors and find that the constraints are satisfied: $E \cap B_r = (V, I) = (2, 1)$, which is a proper subset of $C_r = (IV \leq 8)$. Therefore, the resulting behavior is $E \cap B_r = \{(V, I) = (2, 1)\}$.

This example also shows that constraint-behavior contracts can be applied to environments with different controlled ports.

Although the definition precisely specifies the behaviors of physical components in a compact form, the use of the ITE operator and the exclusion of the empty set limit its applicability. To illustrate this, we first derive the refinement relation for the constraint-behavior contracts:

Definition 4.3. *Given two constraint-behavior contracts $\mathcal{C}_1^{cb} = (C_1, B_1)$ and $\mathcal{C}_2^{cb} = (C_2, B_2)$, $\mathcal{C}_2^{cb}$ is a refinement of $\mathcal{C}_1^{cb}$, denoted as $\mathcal{C}_1^{cb} \succeq \mathcal{C}_2^{cb}$, if they satisfy the following relation:*

$$C_1 \cup \overline{B_1} \subseteq C_2 \cup \overline{B_2}, B_2 \subseteq B_1 \text{ and } B_2 \cap C_1 = B_1 \cap C_1.$$

Proof. First, for all environments such that $\mathcal{C}_1$ exhibits its intrinsic behaviors, $\mathcal{C}_2$ must also exhibit its intrinsic behavior. Given that $\emptyset \subset E \cap B_1 \subseteq C_1$, we derive $E \subseteq C_1 \cup \overline{B_1}$ and $E \cap B_1 \neq \emptyset$. For the contract $\mathcal{C}_2$ to exhibit its intrinsic behavior, we require $E \cap B_2 \subseteq (C_1 \cup \overline{B_1} \cap B_2)$. Thus we obtain $(C_1 \cup \overline{B_1}) \subseteq (C_2 \cup \overline{B_2})$.

Then, for these environments, the resulting behaviors of contracts $\mathcal{C}_2$ must be a subset of those of contracts $\mathcal{C}_1$, requiring $E \cap B_2 \subseteq E \cap B_1$. Using the environment conditions above, we derive $E \cap B_2 \subseteq (C_1 \cup \overline{B_1}) \cap B_2 = C_1 \cap B_2 \cup \overline{B_1} \cap B_2$. To establish the subset relation $C_1 \cap B_2 \cup \overline{B_1} \cap B_2 \subseteq E \cap B_1$, we must have $\overline{B_1} \cap B_2 = \emptyset$ since it cannot be a subset of B_1. Therefore, we obtain $B_2 \subseteq B_1$.

Finally, we need to ensure that these environments, when intersecting with the intrinsic behaviors of $\mathcal{C}_2$, do not lead to an empty set. Given that $E \subseteq C_1 \cup \overline{B_1}$ and $E \cap B_1 \neq \emptyset$, we infer that E must contain an element of $B_1 \cap C_1$. Now, consider the case when $E = B_1 \cap \overline{B_2} \cap C_1$. In this situation, $E \cap B_2 = \emptyset$, violating the environment's requirements. Therefore, $B_1 \cap \overline{B_2} \cap C_1$ should not be a valid environment for $\mathcal{C}_1$, meaning $B_1 \cap \overline{B_2} \cap C_1 = \emptyset$, and thus $B_2 \supseteq B_1 \cap C_1$. Combining it with the relation $B_2 \subseteq B_1$, we conclude that $B_2 \cap C_1 = B_1 \cap C_1$. This constraint ensures that E contains an element of $B_2 \cap C_1$, guaranteeing that $E \cap B_2$ is a non-empty set. □

The result requires that the intrinsic behaviors B_1 and B_2 be identical under the conditions specified by C_1, which restricts the flexibility to refine specifications. This strict relationship limits the ability to refine specifications by reducing the flexibility in behaviors. For example, a specification might allow resistors with resistance R between 1 and 3, and a resistor with $R = 2$ should satisfy this specification through refinement relation. However, the current refinement relation fails to capture this flexibility because the constraint-behavior contract considers any environment satisfying $\emptyset \subset E \cap B \subseteq C$. If an adversarial environment only permits $R = 3$, it forces the resistance to be 3, causing a resistor with $R = 2$ to fail in that environment.

To address this issue and extend the applicability of constraint-behavior contracts, we focus on the desired environment while ignoring adversarial ones. We assume that the environment of interest never leads to unconstrained behaviors (the "else" part of the ITE operator in constraint-behavior contracts) for any contracts. This assumption allows us to

relax constraint checking during contract operations. Consequently, we define the environments as those that satisfy the following two environment axioms:

1. The environment never results in an empty set when the intrinsic behavior is applied $E \cap B \neq \emptyset$.

2. The environment never violates the constraints when the intrinsic behavior is applied $E \cap B \subseteq C$.

For any environments of interest, we must verify that they satisfy the axioms to ensure that the contracts possess the desired semantics. If either axiom is violated, the component may exhibit universal behavior since the constraints would no longer hold. The first axiom addresses the limitation imposed by the empty set requirement, while the second axiom allows us to eliminate the ITE operator, facilitating easier manipulation of the contracts. We will demonstrate its usefulness in the following sections. In the remainder of this chapter, we will assume the above two environment axioms unless explicitly stated otherwise.

4.3 Constraint-Behavior Contracts with Environment Axioms

In this section, we discuss the properties, operations, and relations of constraint-behavior contracts under the two axioms.

First, we can express constraint-behavior contracts as assume-guarantee contracts:

Proposition 4.1. *A constraint-behavior contract $\mathcal{C}^{cb} = (C, B)$ possesses the same semantics as the assume-guarantee contract $\mathcal{C}^{ag} = (C \cup \overline{B}, B)$.*

The equivalence of semantics can be shown by using the contract definitions introduced in 2.2. The environment set $\mathcal{E}$ of a constraint-behavior contract consists of any components that satisfy the constraints when applied to the intrinsic behaviors. Therefore, $\mathcal{E} = \{E \mid E \cap B \subseteq C\} = 2^{C \cup \overline{B}}$. The implementation set $\mathcal{I}$ of constraint-behavior contracts should satisfy the $\mathcal{I} = \{M \mid \forall E \in \mathcal{E},\ M \cap E \subseteq B\}$, which simplifies to $\mathcal{I} = 2^B$. Since an assume-guarantee contract $\mathcal{C}^{ag} = (A, G)$ has an environment set $\mathcal{E} = 2^A$ and implementations $\mathcal{I} = 2^{G \cup \overline{A}}$, we can match the expressions from both semantics to obtain $A = C \cup \overline{B}$ and $G = B$. Note that the two axioms are essential for the derivation to hold, as they remove the need to check for an empty set and disregard unconstrained behaviors resulting from violating constraint checks. Although the assumption A of the corresponding assume-guarantee contract is written as $C \cup \overline{B}$, the environment must still satisfy axioms 1 and 2 to maintain consistent semantics. For example, environments like $E = \overline{B}$ or $E = \mathcal{B}$ should not be applied, as they violate the axioms and lead to unconstrained behavior in the constraint-behavior contract. In the case of $E = \overline{B}$, the constraint-behavior contract

allows unconstrained behavior, whereas the assume-guarantee contract disallows any behaviors. Similarly, for $E = \mathcal{B}$, the constraint-behavior contract permits all resulting behaviors, while the assume-guarantee contract restricts behaviors to the set B.

The following example shows the corresponding assume-guarantee for the resistor in Figure 4.3 (a):

Example 4.2. *The assume-guarantee contract for the constraint-behavior contract $\mathcal{C}_r^{cb}$, according to Proposition 4.1, is $\mathcal{C}_r^{ag} = (IV \leq 8 \vee V \neq 2I, V = 2I)$.*

We observe that all behaviors of $V = 2$ satisfy the assumption, as $(V, I) = (2, 1)$ satisfies $IV \leq 8$, and all other behaviors with $I \neq 1$ satisfy $V \neq 2I$. Therefore, the guarantee is always enforced, and it eliminates all the behaviors satisfying $V \neq 2I$. The only remaining behavior is $(V, I) = (2, 1)$. A similar derivation can be obtained with the environment $I = 1$.

The example also highlights the counter-intuitive nature of encoding assume-guarantee contracts for physical components. In the derived assume-guarantee contract, all behaviors that violate the guarantee are initially accepted based on the assumption but later eliminated by the guarantee. The mixture of constraints and intrinsic behaviors complicates understanding the specification's intention, while the numerous illegal and subsequently discarded behaviors further complicate behavior derivation. Therefore, using constraint-behavior contracts allows designers to write specifications in an intuitive way by preserving the physical meaning of the components without considering the intermediate illegal behaviors.

4.3.1 Operations and Relations

As introduced in Section 2.2, the contract operations and relation can facilitate system reasoning at the specification level. To this end, we discuss the contract operation for constraint-behavior contracts.

4.3.1.1 Composition

First, we define the composition of constraint-behavior contracts as follows:

Definition 4.4. *The composition of two constraint-behavior contracts $\mathcal{C}_1^{cb} = (C_1, B_1)$, $\mathcal{C}_2^{cb} = (C_2, B_2)$, denoted by $\mathcal{C}_1^{cb} \parallel_{cb} \mathcal{C}_2^{cb}$, is a constraint-behavior contract $\mathcal{C}_{12}^{cb} = (C_{12}, B_{12})$, where*

$$C_{12} = C_1 \cap C_2 \text{ and } B_{12} = B_1 \cap B_2.$$

The intuition of the composition operation is that the constraints of both contracts should be satisfied at the same time, and the intrinsic behaviors of both components are in force simultaneously.

We can show that the composition operation for constraint-behavior contracts aligns with the composition of assume-guarantee contracts. Considering the corresponding assume-guarantee contracts $\mathcal{C}_1^{ag} = (A_1, G_1) = (C_1 \cup \overline{B_1}, B_1)$ and $\mathcal{C}_2^{ag} = (A_2, G_2) = (C_2 \cup \overline{B_2}, B_2)$,

we can use assume-guarantee contract composition to show that the composition result is equivalent to the one obtained following Definition 4.4:

$$\begin{aligned}\mathcal{C}_{12}^{ag} &= \mathcal{C}_1^{ag} \parallel \mathcal{C}_2^{ag} \\ &= ((A_1 \cap A_2) \cup \overline{G_1} \cup \overline{G_2}, G_1 \cap G_2) \\ &= (((C_1 \cup \overline{B_1}) \cap (C_2 \cup \overline{B_2})) \cup \overline{B_1} \cup \overline{B_2}, B_1 \cap B_2) \\ &= ((C_1 \cap C_2) \cup \overline{(B_1 \cap B_2)}, B_1 \cap B_2) \\ &= (C_{12} \cup \overline{B_{12}}, B_{12}),\end{aligned}$$

which is the corresponding assume-guarantee contract of the composed constraint-behavior contract $\mathcal{C}_{12}^{cb}$.

The following example illustrates the composition of two resistor specifications when the resistors are in parallel:

Example 4.3. *Consider the two contracts $\mathcal{C}_1^{cb} = (C_{r1}, B_{r1})$ and $\mathcal{C}_2^{cb} = (C_{r2}, B_{r2})$ for two resistors, where $C_{r1} : I_1 V \leq 8$, $B_{r1} : V = 2I_1$, $C_{r2} : I_2 V \leq 16$, and $B_{r2} : V = 4I_2$. The composition is $\mathcal{C}_{12}^{cb} = (C_{r12}, B_{r12})$, where C_{r12} is $(I_1 V \leq 8) \wedge (I_2 V \leq 16)$ and B_{r12} is $(V = 2I_1) \wedge (V = 4I_2)$.*

When the environment provides the voltage $V = 4$, the resulting behavior is $(V, I_1, I_2) = (4, 2, 1)$.

4.3.1.2 Refinement

We define the refinement relation of constraint-behavior contracts:

Definition 4.5. *Given two constraint-behavior contracts $\mathcal{C}_1^{cb} = (C_1, B_1)$ and $\mathcal{C}_2^{cb} = (C_2, B_2)$, $\mathcal{C}_2^{cb}$ is a refinement of $\mathcal{C}_1^{cb}$, denoted as $\mathcal{C}_1^{cb} \succeq \mathcal{C}_2^{cb}$, if they satisfy the following relation:*

$$C_1 \cup \overline{B_1} \subseteq C_2 \cup \overline{B_2} \text{ and } B_2 \subseteq B_1.$$

Similar to composition, we can show that the notion of refinement for constraint-behavior contracts aligns with assume-guarantee contract refinement. Consider their corresponding assume-guarantee contracts $\mathcal{C}_1^{ag} = (A_1, G_1) = (C_1 \cup \overline{B_1}, B_1)$ and $\mathcal{C}_2^{ag} = (A_2, G_2) = (C_2 \cup \overline{B_2}, B_2)$ and suppose they satisfy the refinement relation $\mathcal{C}_1^{ag} \succeq \mathcal{C}_2^{ag}$. Using assume-guarantee contract refinement as in Section 2.2, the refinement relation requires that the following conditions hold:

$$C_1 \cup \overline{B_1} = A_1 \subseteq A_2 = C_2 \cup \overline{B_2}, B_2 = G_2 \subseteq G_1 = B_1$$

which is the same condition as the condition for constraint-behavior contract refinement.

Here we show an example of the contract refinement based on Example 4.3:

Example 4.4. *We want to check if the composition result in Example 4.3 is a refinement of the system contract* $\mathcal{C}_{r3}^{cb} = (C_{r3}, B_{r3}) = ((I_1 + I_2)V \leq 12, V = \frac{4}{3}(I_1 + I_2))$.

First, we check if the intrinsic behaviors satisfy the condition $B_{r12} \subseteq B_{r3}$. *By denoting any element in* B_{r12} *as* $(V, I_1, I_2) = (V, \frac{1}{2}V, \frac{1}{4}V)$, *the element must also be an element in* B_{r3} *since* $V = \frac{4}{3}(\frac{1}{2}V + \frac{1}{4}V)$. *Therefore, we get* $B_{r12} \subseteq B_{r3}$. *This derivation is equivalent to the derivation of the equivalent resistance for parallel resistors:* ${r_3}^{-1} = {r_1}^{-1} + {r_2}^{-1}$, *where* $r_3 = \frac{4}{3}$, $r_1 = 2$, *and* $r_2 = 4$ *are the resistances of the resistors.*

Then we check if the constraints satisfy the relation in Definition 4.5. We first get $C_{r12} \cup \overline{B_{r12}} = (I_1V \leq 8) \wedge (I_2V \leq 16) \vee (V \neq 2I_1) \vee (V \neq 4I_2)$. *An element in* $C_{r3} = (I_1 + I_2)V \leq 12$ *is an element of* C_{r12} *if* $V \neq 2I_1$ *or* $V \neq 4I_2$. *Therefore, we only need to check if all elements satisfying* $V = 2I_1$ *and* $V = 4I_2$ *in* C_{r3} *are elements of* $(I_1V \leq 8) \wedge (I_2V \leq 16)$. *Using the relation between voltages and currents, we know the ratio between* I_1V *and* I_2V *is always* 2. *Therefore, from* $(I_1 + I_2)V \leq 12$, *the maximum value of* I_1V *is* 8, *and the maximum value of* I_2V *is* 4, *which satisfies* $(I_1V \leq 8) \wedge (I_2V \leq 16)$, *and thus* $C_{r3} \subseteq C_{r12} \cup \overline{B_{r12}}$.

Then, as $B_{r12} \subseteq B_{r3}$, *we get* $\overline{B_{r3}} \subseteq \overline{B_{r12}}$ *and* $\overline{B_{r3}} \subseteq C_{r12} \cup \overline{B_{r12}}$. *Combining the results, we get* $C_{r3} \cup \overline{B_{r3}} \subseteq C_{r12} \cup \overline{B_{r12}}$, *which means the refinement relation holds.*

The refinement relation in the example shows that the composed resistor has the same equivalent resistance but a larger range of working regions than the system specification. As a result, the implementation based on the refinement result never fails if the environment always satisfies the specification of B_3. In this example, we can intuitively understand the intrinsic behaviors using the parallel resistance, but not for the constraint, i.e., the maximum power. If the system $r3$ requires a higher maximum power than 12, the refinement does not hold, as the maximum power constraint of r_1 could be violated.

4.3.1.3 Merging

Following the definition of the assume-guarantee contract, we can define the merging of constraint-behavior contracts:

Definition 4.6. *The merging of two constraint-behavior contracts* $\mathcal{C}_1^{cb} = (C_1, B_1)$, $\mathcal{C}_2^{cb} = (C_2, B_2)$, *denoted by* $\mathcal{C}_1^{cb} \cdot \mathcal{C}_2^{cb}$, *is a constraint-behavior contract* $\mathcal{C}_{12}^{cb} = (C_{12}, B_{12})$, *where*

$$C_{12} = (C_1 \cup \overline{B_1}) \cap (C_2 \cup \overline{B_2}) \text{ and}$$
$$B_{12} = B_1 \cap B_2 \cup (B_1 \cap \overline{C_1}) \cup (B_2 \cap \overline{C_2}),$$

which are derived from directly applying the corresponding assume-guarantee contracts in the assume-guarantee contract merging operator.

4.3.1.4 Conjunction

Although the above operations are elegantly defined through the close connection with assume-guarantee contracts, there is a pitfall with the conjunction operator due to the environment axioms. Ideally, we would expect the conjunction operator to combine component

models under different operating conditions, each corresponding to a set of intrinsic behaviors.

Definition 4.7. *(Overly-constrained Conjunction) The conjunction of the assume-guarantee contracts corresponding to the two constraint-behavior contracts* $\mathcal{C}_1^{cb} = (C_1, B_1)$, $\mathcal{C}_2^{cb} = (C_2, B_2)$, *denoted by* $\mathcal{C}_1^{cb} \wedge \mathcal{C}_2^{cb}$, *results in a constraint-behavior contract* $\mathcal{C}_{12}^{cb} = (C_{12}, B_{12})$, *where*

$$C_{12} = (C_1 \cup \overline{B_1}) \cap (C_2 \cup \overline{B_2}) \text{ and}$$
$$B_{12} = B_1 \cap B_2.$$

However, the conjunction using the corresponding assume-guarantee contract falls short of this goal, as it produces a more restricted contract where both intrinsic behaviors must exhibit simultaneously ($B_1 \cap B_2$), rather than allowing the intrinsic behaviors to vary based on different conditions.

The issue arises because incorporating different conditions for a component requires correct semantics: the component should remain unconstrained when the environment is outside the conditions specified by the contracts. This aspect is missing in the corresponding assume-guarantee contracts, as the environment axioms are violated. As a result, the direct application of assume-guarantee contract operations leads to an overly constrained contract, which does not align with our intended behavior for conjunction. The resulting intrinsic behavior, $B_1 \cap B_2$, matches our intention only when $C_1 \cap C_2$ is satisfied, as this is the sole condition where the axioms hold.

Therefore, the conjunction operator for constraint-behavior contracts, when applied under the environment axioms, becomes ineffective for system design reasoning.

4.4 Specifying Component by Combining Multiple Models

In the previous section, we observed that the conjunction operators, when used with the environment axioms, fail to incorporate different models of conditions. This limitation stems from the inherent structure of constraint-behavior contracts, where conditions are not explicitly represented. Instead, constraints are verified after behaviors are applied, rather than being contingent on specific conditions.

A designer may still want to break down complex component models into separate contracts for ease of management. In this section, we propose a new operator to capture this idea within constraint-behavior contracts.

Here we present the operator, *Model Extension*, parametrized by a set B_{ex}.

Definition 4.8. *The modeling extension of two constraint-behavior contracts* $\mathcal{C}_1^{cb} = (C_1, B_1)$, $\mathcal{C}_2^{cb} = (C_2, B_2)$ *by the extension behavior* B_{ex}, *denoted by extension*$(\mathcal{C}_1^{cb}, \mathcal{C}_2^{cb}, B_{ex})$, *is a*

constraint-behavior contract $\mathcal{C}_{12}^{cb} = (C_{12}, B_{12})$*, where*

$$C_{12} = (C_1 \cup C_2) \text{ and}$$
$$B_{12} = (C_1 \cap \overline{C_2} \cap B_1) \cup (\overline{C_1} \cap C_2 \cap B_2) \cup (C_1 \cap C_2 \cap B_1 \cap B_2) \cup (\overline{C_1} \cap \overline{C_2} \cap B_{ex}).$$

There are several important considerations when defining this operator:

1. This operator is not well-defined unless information about intrinsic behavior outside all conditions is provided. Constraint-behavior contracts rely on intrinsic behaviors for checking constraints, which are crucial for maintaining the correct semantics. However, when combining multiple models, the resulting intrinsic behaviors outside their respective conditions depend on information not included in the contracts. Arbitrarily defining these behaviors can still meet the goal but with different semantics. To address this, we introduce an additional set, the extension behaviors B_{ex}, to represent intrinsic behaviors outside all conditions. The designer must provide this set if the conditions do not cover the universe of behavior to ensure the semantics align with the physical rules of the component.

2. To satisfy the environment axioms, the environment only needs to meet the one for the resulting contract: $\emptyset \neq E \cap B \subseteq C_1 \cup C_2$, as the operator has absorbed the axioms required for satisfying the operand contracts. This ensures that the resulting behaviors are consistent with the intended semantics.

The intrinsic behaviors can be understood as selecting the appropriate behavior based on which constraints are satisfied. The resulting intrinsic behaviors are divided into four regions, depending on whether C_1 and C_2 are satisfied after applying their corresponding intrinsic behaviors in the operand contracts. If C_1 is satisfied after applying B_1 while C_2 is violated after applying B_2, the model represented by contract $\mathcal{C}_1$ is activated, and vice versa when C_2 is satisfied after applying B_2 and C_1 is violated after applying B_1. When both constraints are satisfied after applying $B_1 \cap B_2$, both models are activated. The extension behavior B_{ex} explicitly defines what intrinsic behaviors should be applied outside the regions of C_1 and C_2."

A special case is when $C_1 \cup C_2 = \mathcal{B}$ and $C_1 \cap C_2 = \emptyset$. In this case, the operator is well-defined without the extension behaviors, and a model is uniquely selected based on the environment condition.

Here we demonstrate the use of the model extension operator:

Example 4.5. *Diodes are common elements in circuit design, used to control the current direction and stabilize the voltage. A simplified model, widely used in circuit analysis, considers two operating conditions for a diode. First, the diode is in the 'On' state when the voltage across it is at least* V_D*, the threshold voltage that activates the diode. In this state, the voltage remains fixed at* V_D*, allowing any positive current to flow through. Second, the diode is in the 'Off' state when the voltage across it is below* V_D*, at which point no current flows*

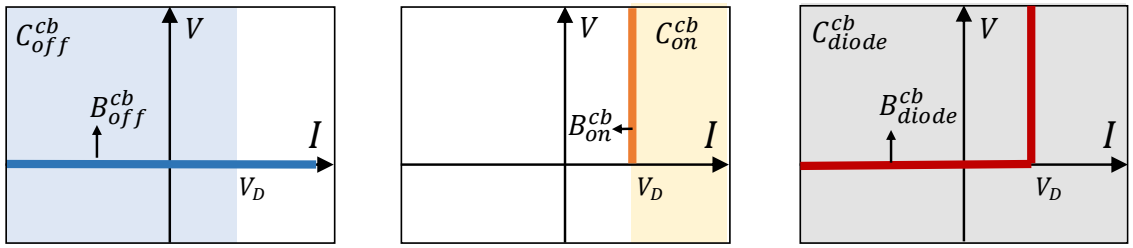


Figure 4.4: Example of model extension on a simplified diode. (a) the illustration of the contract for "Off" condition. (b) the illustration of the contract for "On" condition. (c) the resulting contract after model extension.

through the diode." According to the model, we can derive two constraint-behavior contracts, each corresponding to one state condition:

$$\mathcal{C}^{cb}_{on} : (V >= V_D, V = V_D \wedge I >= 0), \text{ and}$$
$$\mathcal{C}^{cb}_{off} : (V < V_D, I = 0),$$

where $\mathcal{C}^{cb}_{on}$ describes the conditions when the diode is "On", while $\mathcal{C}^{cb}_{off}$ specifies the behaviors when the diode is "Off".

We can apply the model extension operator to combine the two models. As $C_1 \cup C_2 = \mathcal{B}$, the operations can be performed without extension behaviors:

$$\mathcal{C}^{cb}_{diode} = extension(\mathcal{C}^{cb}_{on}, \mathcal{C}^{cb}_{off}, \cdot) = (True, (V >= V_D \wedge V = V_D \wedge I >= 0) \vee (V < V_D \wedge I = 0)).$$

Figure 4.4 visualizes the resulting contracts from the example diode models. The model extension operator ensures that the correct model is selected based on the relevant conditions. With this operator, designers can define separate models for different conditions, allowing for compact and intuitive representations of each model. Additionally, it simplifies model management by delegating the combination of models to contract operations rather than relying on manual effort.

4.5 Constraint-Behavior Contracts and Assume-Guarantee contracts

Constraint-behavior contracts and assume-guarantee contracts have different semantics and usage while they share some similarities in their forms and operations. As introduced in Proposition 4.1, every constraint-behavior contract has an equivalent assume-guarantee contract. Furthermore, the refinement relation and the composition operation of constraint-behavior contracts can be derived from assume-guarantee contracts, though having a slightly

different form. This section discusses the similarities and differences between constraint-behavior contracts and assume-guarantee contracts.

4.5.1 Semantic and Practical Usage

Both constraint-behavior contracts and assume-guarantee contracts are defined as a pair of behavior sets $2^{\mathcal{B}_\mathcal{P}} \times 2^{\mathcal{B}_\mathcal{P}}$. However, the two contracts have to be understood and used differently in practical applications for the same behavior sets. In assume-guarantee contracts, the assumption directly specifies the environment behaviors where the component is expected to function, and the guarantee states the component's responsibility in those environments. The assumption checks the working conditions independently of the contract's guarantee. On the other hand, constraint-behavior contracts describe the relationship between ports through intrinsic behaviors, with constraints indicating the conditions under which the established relationship no longer holds. In this case, both the intrinsic behaviors and constraints are involved to derive the uncontrolled part behavior and check the satisfaction of the component's working conditions.

The semantics define different orders for applying the behavior sets, which results in different practical usage. As shown in Section 4.2.1, the implicit port directions from the interpretations of the assume-guarantee contracts make it hard and counter-intuitive to specify physical components.

We refer to these two interpretations of the behavior sets as the assume-guarantee contract semantics and the constraint-behavior contract semantics.

Besides the order for applying the behavior sets, the two contract formalisms also differ in how they treat environment behaviors.

For assume-guarantee contracts, given an environment's behaviors E, the resulting behaviors are:

$$\begin{aligned} E \cap (G \cup \overline{A}) &= (E_1 \cup E_2) \cap (G \cup \overline{A}) \\ &= (E_1 \cap (G \cup \overline{A})) \cup (E_2 \cap (G \cup \overline{A})) \\ &= \bigcup_{e \in E} \{e\} \cap G \cup \overline{A} \end{aligned}$$

This definition allows the environment to be decomposed into its subsets and even elements. Each subset of E independently contributes to the resulting behaviors, which are combined through a set union operation. If one subset of environment behaviors (E_1) violates the assumptions while another subset E_2 satisfies the assumptions, the system still guarantees $E_2 \cap G$ under E_2, as if it were operating in a valid environment. Let's say E_1 are those behaviors causing the violation while another assumption, $E_2 \cap (G \cup \overline{A}) = E_2 \cap G$, which still produces behaviors as if the system is working under the targeted environment.

In contrast, constraint-behavior contracts evaluate the entire environment's behaviors collectively, and any violation of the constraints would lead to a total failure for any environment behaviors. For example, in the contract from Example 4.1, if the environment behaviors

are $V = 4$, the resulting behavior is $(V, I) = (4, 2)$, as the constraint is satisfied. However, if the environment behaviors are $V = 4 \vee V = 5$, the constraint is violated $E \cap B \not\subseteq C$, and the resulting behaviors are the entire E. As a result, behaviors like $(V, I) = (4, 5)$, which are contributed by $V = 4$ instead of the cause of the violation $V = 5$, are included. This difference highlights a key distinction between the two formalisms: while assume-guarantee contracts isolate violations to specific elements of the environment, constraint-behavior contracts treat any violation by one element as a failure to produce correct behaviors for all elements in the environment.

4.5.2 Equivalent Saturated Form

According to the refinement relation, we can define the saturation of a constraint-behavior contract to reason about the partial order between the contracts:

Definition 4.9. *The saturation of a constraint-behavior contract, denoted by the operator $sat_{cb}()$, is defined as*

$$sat_{cb}(C, B) = (C \cup \overline{B}, B).$$

The saturation results in a maximal contract, based on partial order defined by refinement, which represents the same specification.

A saturated constraint-behavior contract denotes the same specification as the saturated assume-guarantee contract of the same form, meaning that the interpretation results of the two semantics are common for saturated contracts. We can denote a saturated constraint-behavior contract as $\mathcal{C}^{cb} = (C, B) = (S_1, S_2)$ and a saturated assume-guarantee contract $\mathcal{C}^{ag} = (A, G) = (S_1, S_2)$, where S_1 and S_2 are behavior sets.

In the following, we show their equivalence. First, we show that any pair of behavior sets (S_1, S_2) being a saturated constraint-behavior contract is also a saturated assume-guarantee contract under the assume-guarantee contract semantics, and vice versa:

Lemma 4.1. *Given a pair of behavior sets $(S_1, S_2) \in 2^{\mathcal{B}_\mathcal{P}} \times 2^{\mathcal{B}_\mathcal{P}}$, $S_2 = S_2 \cup \overline{S_1}$ if and only if $S_1 \cup \overline{S_2} = S_1$.*

Proof. $S_2 \cup \overline{S_1} = S_2 \Leftrightarrow \overline{S_1} \subseteq S_2 \Leftrightarrow \overline{S_2} \subseteq S_1 \Leftrightarrow S_1 \cup \overline{S_2} = S_1$. □

Therefore, any saturated constraint-behavior contract is also a saturated assume-guarantee contract if we interpret its pair of behavior sets in the assume-guarantee contract semantics.

Then we show that the saturated contracts in the two semantics represent the same specification using the mathematical meta-theory of contracts [17].

Proposition 4.2. *Given a pair of behavior sets $(S_1, S_2) \in 2^{\mathcal{B}_\mathcal{P}} \times 2^{\mathcal{B}_\mathcal{P}}$, if $S_2 = S_2 \cup \overline{S_1}$, then the pair expresses the same specification under the constraint-behavior contract semantics and the assume-guarantee contract semantic, i.e., specifying the same sets of environments and implementations.*

Proof. First, we show that the two semantics result in the same environment set: $\mathcal{E}^{cb} = \{E \in \mathcal{B}_{\mathcal{P}} \mid E \cap S_2 \subseteq S_1\} = \left\{E \;\middle|\; E \subseteq S_1 \cup \overline{S_2} = S_1\right\} = 2^{S_1} = \mathcal{E}^{ag}$.

Then we show that the two semantics represent the same implementation set: $\mathcal{I}^{cb} = \left\{I \subseteq \mathcal{B}_V \;\middle|\; \forall E \in \mathcal{E}^{cb}, I \cap E \subseteq S_2\right\}$
$= \{I \subseteq \mathcal{B}_V \mid I \cap S_1 \subseteq S_2\} = \left\{I \subseteq \mathcal{B}_V \;\middle|\; I \subseteq S_2 \cup \overline{S_1}\right\} = 2^{S_2 \cup \overline{S_1}} = \mathcal{I}^{ag}$, where the second equality is obtained by $E \subseteq S_1$ since $\mathcal{E}^{cb} = 2^{S_1}$. Therefore, the two semantics on the pair of behavior sets express the same specification since they specify identical environment and implementation sets. □

This property allows us to derive the formula for operations of constraint-behavior contracts, like the derivation in Section 4.2.2. Furthermore, algorithms based on saturated assume-guarantee contracts can be applied to saturated constraint-behavior contracts, allowing integration of the two contract semantics.

4.5.3 Unsaturated Composition with Set Intersection

The previous parts detail the close relationship between assume-guarantee contracts and constraint-behavior contracts. However, the constraint-behavior contracts have a property that the assume-guarantee contracts do not have: the same composition formula for saturated and unsaturated constraint-behavior contracts.

The composition in Definition 4.4, which involves only simple set intersections, does not require the contracts to be saturated, and the resulting contract might also be unsaturated. However, assume-guarantee contracts do not possess a similar property. The composition formula introduced in Section 2.2 cannot be applied to unsaturated assume-guarantee contracts. Consider unsaturated contracts $\mathcal{C}_1^{ag} = (A_1, G_1)$ and $\mathcal{C}_2^{ag} = (A_2, G_2)$, where A_1, G_1, A_2, and G_2 are unconstrained sets. We can obtain their composition as follows:

$$\begin{aligned}
\mathcal{C}_1^{ag} \parallel \mathcal{C}_2^{ag} &= \mathrm{sat}_{\mathrm{ag}}(\mathcal{C}_1^{ag}) \parallel \mathrm{sat}_{\mathrm{ag}}(\mathcal{C}_2^{ag}) \\
&= (A_1, G_1 \cup \overline{A_1}) \parallel (A_2, G_2 \cup \overline{A_2}) \\
&= ((A_1 \cap A_2) \cup (\overline{G_1} \cap A_1) \cup (\overline{G_2} \cap A_2), \\
&\quad (G_1 \cup \overline{A_1}) \cap (G_2 \cup \overline{A_2})).
\end{aligned}$$

Note that no further simplifications can be made since all sets are unconstrained. Observing the obtained composition formula, the composition of unsaturated assume-guarantee contracts still involves the saturation operation such as $G_1 \cup \overline{A_1}$ and $G_2 \cup \overline{A_2}$. On the other hand, constraint-behavior contracts have a common composition formula for saturated and unsaturated contracts. Considering two unsaturated contracts $\mathcal{C}_1^{cb} = (C_1, B_1)$ and $\mathcal{C}_2^{cb} = (C_2, B_2)$,

where C_1, B_1, C_2, and B_2 are unconstrained sets, we can obtain their composition as follows:

$$\begin{aligned}\mathcal{C}_1^{cb} \parallel_{cb} \mathcal{C}_2^{cb} &= \mathrm{sat}_{\mathrm{cb}}(\mathcal{C}_1^{cb}) \parallel \mathrm{sat}_{\mathrm{cb}}(\mathcal{C}_2^{cb}) \\ &= (C_1 \cup \overline{B_1}, B_1) \parallel (C_2 \cup \overline{B_2}, B_2) \\ &= (((C_1 \cup \overline{B_1}) \cap (C_2 \cup \overline{B_2})) \cup \overline{B_1} \cup \overline{B_2}, \\ &\quad B_1 \cap B_2) \\ &= (((C_1 \cap C_2) \cup \overline{(B_1 \cap B_2)}, B_1 \cap B_2) \\ &= \mathrm{sat}_{\mathrm{cb}}(\mathcal{C}_1^{cb} \parallel_{cb} \mathcal{C}_2^{cb}),\end{aligned}$$

where we first saturate the constraint-behavior contracts to get their equivalent assume-guarantee contacts and then perform the assume-guarantee contract composition.

As a result, constraint-behavior contracts can be composed simply by set intersection, even though they are unsaturated. This property also demonstrates the ease of use of our proposed constraint-behavior contracts. The composition is straightforward and intuitive for designers to express their intentions and identify potential faults.

4.6 Verification using Constraint-behavior Contracts and Assume-guarantee Contracts

By comparing constraint-behavior and assume-guarantee contracts, we have shown the possibility of integrating contracts from different semantics into the contract-based design process. Based on this, this section introduces a verification methodology that utilizes constraint-behavior contracts and assume-guarantee contracts.

Depending on the application's need, a system may be specified with clear port directions while the underlying components can be expressed using implicit equations. In this case, the designers may choose assume-guarantee contracts to specify the system while utilizing constraint-behavior contracts for its components. Therefore, a verification methodology for such an integration is necessary, as verification of these specifications requires the refinement relation to be established under different contract formalisms.

With Proposition 4.2, we can derive the refinement relation and verify system specification for this case. First, we show the conditions for contract refinement for an assume-guarantee contract and a constraint-behavior contract as follows:

Definition 4.10. *Given a constraint-behavior contract $\mathcal{C}^{cb} = (C, B)$ and an assume-guarantee contract $\mathcal{C}^{ag} = (A, G)$. $\mathcal{C}^{cb}$ refines $\mathcal{C}^{ag}$, denoted as $\mathcal{C}^{cb} \preceq \mathcal{C}^{ag}$, if the following condition is satisfied:*

$$A \cap B \subseteq C \cap G.$$

The intuition behind the definition is to ensure that all intrinsic behaviors under the specified environments ($A \cap B$) result in behaviors satisfying the constraints of the components and the system guarantees ($C \cap G$).

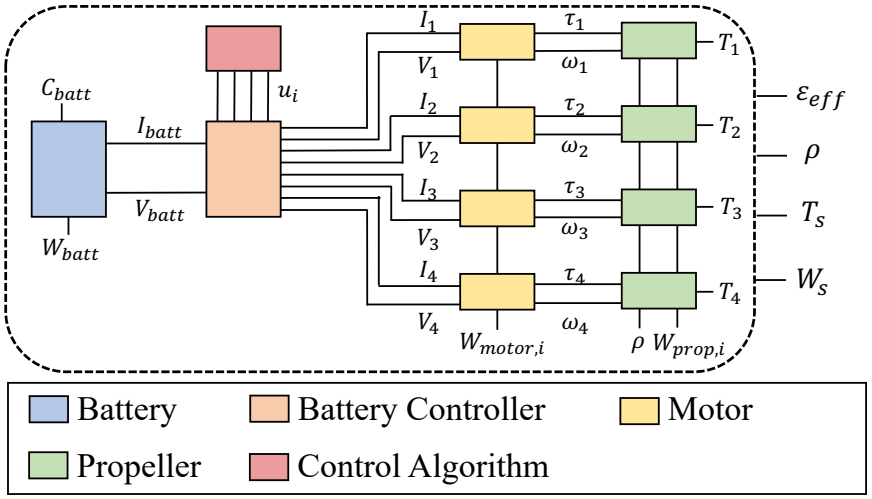


Figure 4.5: System diagram of a UAV propulsion system with four propellers.

We can use Proposition 4.2 to show that the definition aligns with saturated contract refinement. First, we define the assume-guarantee contract $\mathcal{C}_2^{ag} = (C \cup \overline{B}, B)$, which expresses the same specification as $\mathcal{C}^{cb}$. According to assume-guarantee contract refinement, the conditions for contract refinement are $A \subseteq C \cup \overline{B}$ and $B \subseteq G \cup \overline{A}$. By rewriting $A \subseteq C \cup \overline{B}$, we obtain the equivalent condition $A \cap B \subseteq C$. Similarly, rewriting $B \subseteq G \cup \overline{A}$ yields the equivalent condition $A \cap B \subseteq G$. Combining these two results, we get $A \cap B \subseteq C \cap G$, which aligns with the defined refinement relation.

Here we sketch a verification process integrating constraint-behavior contracts and assume-guarantee contracts using the contract refinement as follows: first, the designer specifies the system requirement in assume-guarantee contracts. Then the designer gathers the component specifications written in constraint-behavior contracts and assume-guarantee contracts. After gathering all the contracts, the composed contract is computed by applying contract composition to the component contracts. Finally, the designer can verify the system by checking contract refinement following Definition 4.10.

4.7 Demonstration: UAV Electrical System Design

To demonstrate the effectiveness of constraint-behavior contracts in addressing design problems with physical components, we apply them to a UAV propulsion system verification problem. The system specification includes two requirements: 1) the UAV must tolerate the maximum voltage of the batteries, and 2) the UAV must be able to stay in the air for a given time t_req. The demonstration involves formulating constraint-behavior contracts for the components and utilizing the verification process that combines constraint-behavior contracts with assume-guarantee contracts.

4.7.1 System Details

The UAV propulsion system is a heterogeneous system spanning the electrical, mechanical, and control domains. Figure 4.5 shows the system overview. The components in the UAV propulsion system with N propellers include a battery pack, N motors, N propellers, a battery controller, and a battery control algorithm [171]. Each individual component of the same type may have a different component model, which has the same ports while the behaviors are different based on the parameter values of the component model. In the following, we detail each component.

- *Battery.* The battery provides electrical power to the propulsion system. A battery has four ports: C_{batt}, I_{batt}, V_{batt}, and W_{batt}. Port C_{batt} indicates the maximum capacity. I_{batt} is the battery current. V_{batt} is the battery voltage. W_{batt} is the weight of the battery.

- *Motor.* The motor converts the electrical energy from batteries to mechanical energy. The motor has five ports: V_i, I_i, τ_i, ω_i, and $W_{motor,i}$. The subscript i denotes the i^{th} component in the system. Port V_i is the voltage across the motor. I_i is the current passing through the motor. τ_i is the torque of the motor. ω_i is the angular velocity of the motor. $W_{motor,i}$ is the weight of the motor.

- *Propellers.* The propeller produces thrust, an upward force for the UAV to fly against gravity. A propeller has five ports: τ_i, ω_i, T_i, ρ, and $W_{prop,i}$. Port τ_i is the torque of the propeller. ω_i is the angular velocity of the propeller. T_i is the thrust generated by the propeller. ρ is the air density. $W_{prop,i}$ is the weight of the propeller.

- *Battery Controller.* The battery controller determines the power drawn from the batteries and its distribution to the motors. A battery controller contains $3N + 2$ ports: V_{batt}, I_{batt}, u_i, V_i, and I_i, for all $i = 1$ to N. Port V_{batt} is the voltage from the battery. Port I_{batt} is the current from the battery. Ports u_i are the control inputs indicating the ratio between V_i and V_{batt}. V_i is the electromotive force provided by the controller. I_i are the current sent by the controller.

- *Control Algorithm.* A control algorithm applies a strategy to control the battery controller such that the UAV can fly to achieve the design goals. Ideally, a control algorithm should send the control signal based on the status of the UAV. However, as we do not focus on the control algorithm in this demonstration, the status can be abstracted as any possible values. As a result, the control algorithm contains N ports: u_i for $i = 1$ to N, where u_i is the control value for the voltage to the motor.

4.7.2 Contract Formulation

As outlined in Section 4.6, the designers first specify the system requirements in assume-guarantee contracts and gather the component specifications in constraint-behavior con-

tracts. Here we show the contract formulation for the system requirements and individual components.

4.7.2.1 System Contract

The first system requirement indicates the UAV must be able to fly, i.e., not crash onto the ground, at the maximum battery voltage. Therefore, we can write the assume-guarantee contract $\mathcal{C}^{ag}_{fly} = (A_{fly}, G_{fly})$ as follows:

A_{fly}	G_{fly}
$\rho = 1.225$	$T_s \geq W_s$
$W_s = W_{battery} + W_{body} + \sum_{i=1}^{n}(W_{prop,i} + W_{motor,i})$	
$T_s = \sum_{i=1}^{n} T_i$	
$u_i = 1 \ \ \forall i = 1...n$	

The assumption sets the air density, defines auxiliary variables for total weight and thrust, and then sets the control output to 1 to operate the UAV at the maximum battery voltage. The guarantee requires that the thrust exceeds the weight, preventing the UAV from crashing onto the ground due to insufficient thrust. In the formulation, W_{body} is a parameter for specifying the weight of the UAV frame and its payload.

The second system requirement indicates that the UAV can stay flying for at least t_req seconds before depleting the battery power. Thus, we can formulate the assume-guarantee contract $\mathcal{C}^{ag}_{t\_req} = (A_{t\_req}, G_{t\_req})$ for this requirement as follows:

A_{t_req}	G_{t_req}
$\rho = 1.225$	$I_{batt} \leq \frac{C_{batt}}{t\_req}$
$W_s = W_{battery} + W_{body} + \sum_{i=1}^{n}(W_{prop,i} + W_{motor,i})$	
$T_s = \sum_{i=1}^{n} T_i$	
$T_s = W_s$	

The assumption sets the air density, defines auxiliary variables for total weight and thrust, and then ensures that the thrust is equivalent to the weight to maintain hovering. The guarantee requires that the current withdrawn from the battery can continuously supply power for at least t_{req} seconds before depleting its stored energy.

4.7.2.2 Component Contract

After formulating the assume-guarantee contracts for the system requirements, we proceed to specify the behavior of the components by listing the constraint-behavior contract formulations.

Battery The contract for a battery of the component model b, denoted as $\mathcal{C}^{cb}_{batt,b} = (C_{batt,b}, B_{batt,b})$, is formulated as follows:

$C_{batt,b}$	$B_{batt,b}$
$I_{batt} \leq I_{max,b}$	$C_{batt} = C_b$
	$V_{batt} = V_b$
	$W_{batt} = W_b$

In the above equations, the following parameters are constants specified by the battery model: $I_{max,b}$ is the maximum allowable current, C_b is the capacity, V_b is the voltage, and W_b is the weight of the battery model.

Motors The contract of a motor of the component model m with index i, denoted as $\mathcal{C}^{cb}_{motor,m,i} = (C_{motor,m,i}, B_{motor,m,i})$, is defined as follows:

$C_{motor,m,i}$	$B_{motor,m,i}$
$V_i I_i < P_{max,m}$	$I_i r_{w,m} = V_i - \frac{\omega_i}{K_{v,m}}$
$I_i < I_{max,m}$	$\tau_i = \frac{K_{t,m}}{r_{w,m}}(V_i - r_{w,m} \times I_{idle,m} - \frac{\omega_i}{K_{v,m}})$
	$W_{motor,i} = W_m$

In the above equations, the following parameters are constants specified by the motor model: $P_{max,m}$ is the maximum allowable power, $r_{w,m}$ is the internal resistance, $K_{v,m}$ is the motor velocity constant, $I_{idle,m}$ is the idle current, $K_{t,m}$ is the motor torque constant, and W_m is the weight of the motor model.

Propellers The contract of a propeller of the component model p with index i, denoted as $\mathcal{C}^{cb}_{prop,p,i} = (C_{prop,p,i}, B_{prop,p,i})$, is defined as follows:

In the above equations, the following parameters are constants specified by the propeller model: D_p is the propeller diameter, $C_{pmin,p}$ and $C_{pmax,p}$ define the range of the power coefficient, $C_{tmin,p}$, $C_{tmax,p}$ define the range of the thrust coefficient, and W_p is the weight of the propeller model.

Battery Controller The contract of the battery controller for battery controller model c, denoted as $\mathcal{C}^{cb}_{batcont,c} = (C_{batcont,c}, B_{batcont,c})$, is defined as follows ($\epsilon_{eff,c}$ is the conversion efficiency of the battery controller):

$C_{prop,p,i}$	$B_{prop,p,i}$
True	$\tau_i = \frac{\rho C_{p,i} \omega_i^2 \times D_p^5}{(2\pi)^3}$ $T_i = \frac{\rho C_{t,i} \omega_i^2 \times D_p^4}{(2\pi)^2}$ $\omega_i \geq 0$ $C_{p,i} \in [C_{pmin,p}, C_{pmax,p}]$ $C_{t,i} \in [C_{tmin,p}, C_{tmax,p}]$ $W_{prop,i} = W_p$

$C_{batcont,c}$	$B_{batcont,c}$
$V_{motor,i} \leq V_{battery}\ \forall i = 1...n$	$I_{batt} = \epsilon_{eff,c} \sum_{i=1}^{n} I_i$ $V_i = u_i V_{battery}$

Control Algorithm As we abstract sensors and all states of the UAV, the control output can be any value between 1 and 0. The value denotes the ratio of the battery voltage sent to the motor. For example, a control output of 0.4 and a battery voltage of $22.2V$ means the motor gets $8.88V$. Therefore, the contract of the control algorithm, denoted as $\mathcal{C}_{cont}^{cb} = (C_{cont}, B_{cont})$ is defined as follows:

C_{cont}	B_{cont}
True	$u_i \in [0, 1]\ \forall i = 1...n$

4.7.3 Designs for Verification

The benchmark designs for verification are based on five designs developed under the DARPA SDCPS project [44]. Among the designs, Designs 1 and 3 are manually designed quadcopters, while the remaining designs are randomly generated by specifying a random number of

	#m	#b	Propeller Model	Motor Model	Battery Model
Design1	4	3	apc_propellers_17x6	t_motor_AT4130KV300	TurnigyGraphene6000mAh6S75C
Design2	4	2	apc_propellers_16x6E	t_motor_AT4130KV230	TattuPlus15C16000mAh12S1Pcompact
Design3	4	2	apc_propellers_6x4E	t_motor_AT2312KV1400	TurnigyGraphene1000mAh2S75C
Design4	6	3	apc_propellers_20x10E	t_motor_AT4130KV230	TurnigyGraphene1400mAh4S75C
Design5	4	1	apc_propellers_11x4_6SF	kde_direct_KDE700XF_535_G3	TattuPlus25C22000mAh12S1PAGRI

Table 4.1: The statistics of the benchmark designs, including the number of batteries in the battery pack (#b), the number of motors (#m), and the component models for propellers, motors, and batteries.

	t_req	W_{body}	$\mathcal{C}^{ag}_{fly}$	$\mathcal{C}^{ag}_{t\_req}$	Reason
Design1	200 s	19.62 N	O	O	
Design2	200 s	19.62 N	X	–	P_motor violated
Design3	200 s	19.62 N	X	–	Not enough thrust
Design4	200 s	19.62 N	O	X	Low capacity
Design5	200 s	19.62 N	O	O	

Table 4.2: The requirement parameters (t_req and W_{body}) of the UAV and the verification result. The second contract is denoted as "–" if the fly requirement is not met as there is no need to verify the requirement.

components and assigning random component models. Table 4.1 provides a summary of the statistics of the designs and component models used in each design.

4.7.4 Verification Settings and Results

We implemented the verification process of contracts using the SMT solver Z3 [48] in the Python programming language, with polynomial arithmetic as the background theory. For designs with multiple batteries, we modify the contract by multiplying the capacity and weight by the number of batteries and dividing the maximum current by the number of batteries, assuming a parallel connection of the batteries. The parameter t_{req} was set to $200s$, and W_{body} was set to $19.62N$ for all benchmark designs. The results of the verification, including the parameter settings, verification outcomes, and reasons for not passing the requirements, are summarized in Table 4.2. The reason for not passing is obtained by analyzing the counter-example provided by the solver.

The verification results show that Designs 1 and 5 passed both design requirements, while Designs 2, 3, and 4 violated at least one of the requirements. Design 2 exceeded the maximum power constraint of the motor, resulting in a violation of the guarantee in $\mathcal{C}^{ag}_{fly}$. Design 3 failed to provide sufficient thrust to lift the UAV, thus violating the guarantee of the contract $\mathcal{C}^{ag}_{t\_req}$. Design 4 required a current greater than the specified value and, as a result, failed to provide guarantees in the contract $\mathcal{C}^{ag}_{t\_req}$ to maintain hovering for the required period.

4.7.5 Discussion

The constraint-behavior contracts enable contract formulations using implicit equations without considering the port directions. This prevents the need to solve implicit equations, reduces the size of the contract library, and provides an intuitive encoding.

For example, let's consider the contract $\mathcal{C}^{cb}_{motor,m,i}$. To verify the first requirement, the motor's input should be the voltage V_i since the requirement implies that the control parameters are system inputs, which directly control the motor's voltage. On the other hand,

to verify the second requirement, the motor's inputs should be the ports connected to the propeller, namely ω_i and τ_i, as the requirement indicates that the thrust is a system input.

By using the constraint-behavior contract for motors, these two requirements can be directly verified without the need to solve implicit equations involving ω_i, I_i, and V_i for various inputs. Additionally, it eliminates the need to store different contracts for the same components solely based on different port directions. As a result, the number of contracts is reduced and independent of the combination of port directions. Furthermore, implicit equations enable designers to formulate contracts using their physical intuition, as the component modeling in [171]. This capability helps prevent specification faults and makes it easier for designers to identify mistakes.

Overall, the demonstration shows the contract formulation, verification process, and the benefits of using constraint-behavior contracts over assume-guarantee contracts.

4.8 Conclusion

We have presented constraint-behavior contracts as specifications for physical components in cyber-physical systems. Unlike assume-guarantee contracts, the intuitive implicit equations in the constraint-behavior contract eliminate the need for equation solving and reduce the number of required contracts. With the developed properties, the proposed verification process can integrate specifications in constraint-behavior contracts and assume-guarantee contracts. The demonstration based on the UAV propulsion system design problem has provided examples of contract formulation, the verification process in an actual design problem, and the benefits brought by the capability of implicit equations.

Chapter 5

Verification: Correct Decomposition in Independent Design

As discussed in Chapter 3, ensuring correct decomposition is essential for facilitating independent design in contract-based design methodologies and ensuring the robustness of the contract-based design process. While refinement is commonly used to verify decomposition correctness, relying solely on refinement for assume-guarantee contracts may may result in implementations that fail to operate correctly in the system's targeted environment, thereby impeding independent design. This chapter explored the *vacuous implementation problem*, which highlights the limitations of contract refinement in providing sufficient assurance. To address this, the concept of *contract replaceability*, a binary relation on contracts, is introduced. The relation is further strengthened as *strong replaceability*, a transitive property that ensures contract replaceability. The requirements are further extended to include *receptiveness* as a constraint on assume-guarantee contracts to ensure strong replaceability. The properties derived from the constraint ensure that strong replaceability holds under contract refinement and cascade composition. Furthermore, conditions for ensuring strong replaceability in feedback composition are identified through an analysis of *fixed obligations* and *fixed obligation graphs*, which characterize the relationships among behaviors collaboratively allowed by subsystem contracts. Based on these findings, algorithms leveraging set operations and satisfiability modulo theories (SMT)-based encoding are proposed, avoiding reliance on specific underlying theories in contract descriptions. By addressing this gap in contract-based design methodology, the proposed conditions and algorithms ensure correct contract decomposition, and thus strengthen the robustness of independent design.

5.1 Introduction

As the needs for large-scale systems, such as autonomous driving, industry 4.0, and artificial intelligence-based applications, increased over the last decades, complexity and heterogeneity have become the main challenges that prolong the design process and increase its

cost [149, 151]. Several methodologies and algorithms have been proposed to cope with design complexity and heterogeneity in all design aspects including specification, verification, and synthesis [43, 121, 126, 151]. Among them, design specification is crucial, as it is the first stage in a rigorous design flow. Methodologies for design specification affect efficiency in verification and synthesis, the subsequent stages of a design flow.

Contract-based design [109, 119, 149, 151] tackles complexity and heterogeneity coupled with platform-based design and formal specifications and thus has become a promising candidate for facilitating complex and heterogeneous design. Contracts are formal specifications [17] for the design environment and its implementation. Contract-based design is a methodology that utilizes contracts in platform-based design. It applies refinement and abstraction to reduce complexity and separates orthogonal viewpoints, or aspects, of a design to handle the heterogeneity of the design [20]. Various contract formalisms have been proposed for different systems, including interface input/output automata [94], assume-guarantee contracts [20], and constraint-behavior contracts [181]. Among many formalisms, assume-guarantee contracts, consisting of an assumption set and a guarantee set, are attractive in various CPS application [43, 106, 121, 148, 164] because of their ease of use.

An assume-guarantee contract $\mathcal{C} = (A, G)$ is defined by a pair of behavior sets, where A is the behaviors that the targeted environment should provide, and G represent the behaviors the system should follow under these environment behaviors. Operations and relations for contracts [77, 81, 132] have been proposed to facilitate the creation of subsystem contracts from different design viewpoints and design hierarchies. Among these, the composition operation and refinement relation are crucial for ensuring the satisfaction of the system contract by independent subsystem contracts. The composition operation combines the subsystem contracts into a monolithic one, while the refinement relation determines whether all implementations of the subsystem contracts satisfy the system contract. Specifically, consider two contracts $\mathcal{C}_s = (A_s, G_s)$ and $\mathcal{C}_{comp} = (A_{comp}, G_{comp})$. $\mathcal{C}_s$ represent the system contract, while $\mathcal{C}_{comp}$ is the composition result of the subsystem contracts. We say $\mathcal{C}_{comp}$ refines $\mathcal{C}_1$ if $A_1 \subseteq A_{comp}$ and $G_1 \supseteq G_{comp}$, as the refined contract $\mathcal{C}_{comp}$ apply to at least all environment of A_1 and the resulting behaviors never violate the requirement specified by G_1.

Independent design [17] is a benefit brought by contract-based design. It allows earlier verification of the system and protects the trade secrets between designers and suppliers. In the independent design paradigm, system-level specifications are refined with more detailed information and decomposed into multiple parts where the composition of these parts satisfies the system-level specifications. The refinement and decomposition ensure that the system meets the top-level requirement once each part follows its local specification. Every supplier thus can independently develop the part under its specification without the system-level specifications or coordination between the suppliers. As a result, the paradigm captures design faults at the specification stage to avoid costly and time-consuming redesign processes and protects the high-level design ideas from leaking to the suppliers, which might be different companies.

Figure 5.1 shows the ideal flow of independent design. First, the top-level specification for the product is decomposed into the specifications of multiple subsystems or parts. These

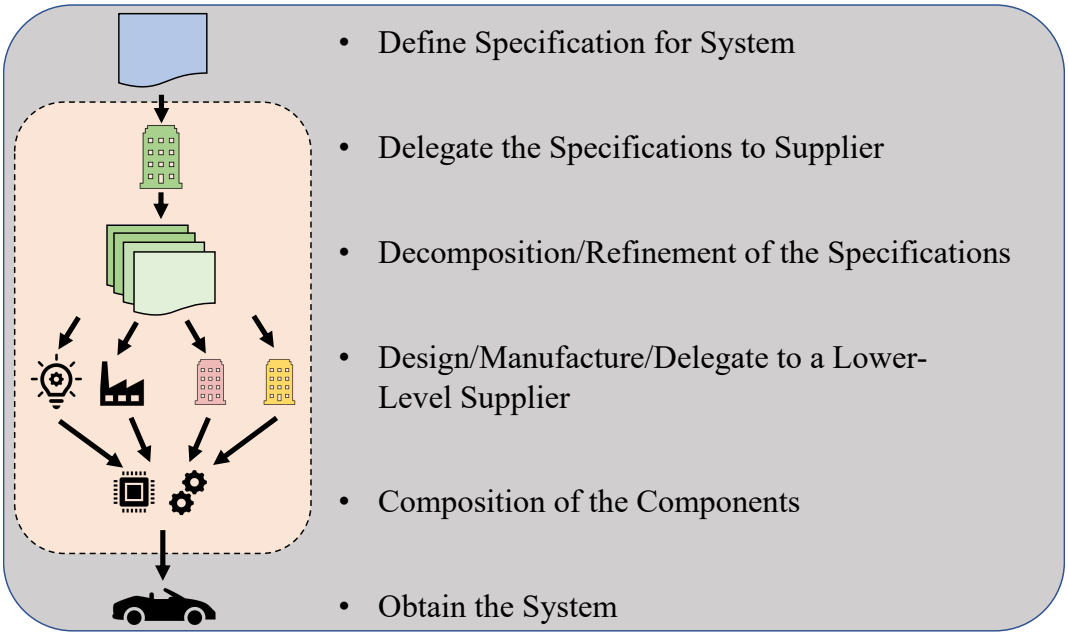


Figure 5.1: Overview of the independent design flow.

specifications of the parts are sent to different suppliers proficient in the domain knowledge to design and provide implementations for the parts. These suppliers can refine their specifications, add more detailed information to the specification, or further decompose the part specification into more part specifications and then delegate them to subsequent suppliers for implementation. After a provider completes an implementation, the implementation is sent back to the system integrator and composed into the target system.

However, the machinery of assume-guarantee contracts does not rule out that the implementations generated for systems or components may not operate correctly in their targeted environments. These implementations, which we call *vacuous implementations*, have empty sets of behaviors in the targeted environment (i.e., not compatible with the environment). Therefore, vacuous implementations should be avoided in the independent design flow. As these vacuous implementations are not excluded from the standard contract framework, additional requirements and constraints enforced on contracts are required to support independent design using assume-guarantee contracts.

In this chapter, we investigate the requirements and additional constraints on assume-guarantee contracts to provably avoid vacuous implementations. Our contributions are the following:

- We identify the vacuous implementation problem as an obstacle to the independent design paradigm. To the best of our knowledge, this is the first work that discovers and addresses this problem of the application of the contract-based design methodology to independent design.

- We then introduce *strong replaceability*, a transitive binary relation, as a restriction on refinement to prevent the vacuous implementation problem in successive refinement steps and enable the independent design. Strong replaceability relieves the need for system contracts to enforce the requirement of replaceability. Once each refinement step follows *strong replaceability*, the resulting contract is guaranteed to contain implementations compatible with the original contract.

- We introduce the concept of *receptiveness* as a property of assume-guarantee contracts. We show that receptiveness is sufficient to ensure strong replaceability for refinement and cascade composition.

- We identify the necessary and sufficient conditions required to ensure strong replaceability for feedback composition based on their corresponding *fixed obligation graphs*. By incorporating receptiveness and strong replaceability, we fully resolve the vacuous implementation issues in assume-guarantee contracts, ensuring the benefits of independent development.

- We propose verification algorithms based on the developed conditions. These algorithms can serve as a foundation for automatic verification for contract-based design.

The remainder of the chapter is organized as follows: Section 5.2 describes the vacuous implementation problem, formulates the contract replaceability requirement, proposes the notion of contract receptiveness, and shows that contract receptiveness ensures strong replaceability in refinement and cascade composition. Section 5.3 presents the necessary and sufficient conditions to ensure strong replaceability in feedback composition and the proposed algorithms. Finally, Section 5.4 concludes the chatper.

5.2 Contract Replaceability for Correct Decomposition and Independent Design

As introduced before, the empty sets of behaviors could be problematic in the set-based definition of contracts. Given a contract $\mathcal{C} = (A, G)$, an implementation of the contract $\mathcal{C}$ is a component $M_{\mathcal{C}}$ such that $M_{\mathcal{C}} \cap A \subseteq G$. Since an empty set is a subset of G, a component $M'_{\mathcal{C}}$ such that $M'_{\mathcal{C}} \cap A = \emptyset$ is by definition an implementation of $\mathcal{C}$. However, this implementation is not compatible with the targeted environment A. We call such an implementation a "vacuous implementation" of the contract. We also define a "strict implementation" of the contract $\mathcal{C}$ as an implementation $M_{\mathcal{C}}$ such that $M_{\mathcal{C}} \cap A \neq \emptyset$.

During the design process, we should avoid vacuous implementations and guarantee strict implementations. However, the refinement of contracts results in smaller acceptable behavior sets, and thus we may lose all strict implementations for the original contracts. Consider a scenario where the contract is $\mathcal{C} = (A, G)$ and its refinement contract is $\mathcal{C}' = (A', G')$ such that the acceptable behavior set of the refined contract and the original assumption set are

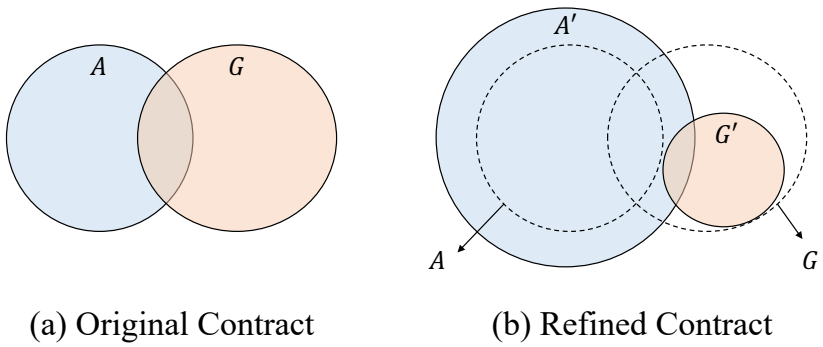


Figure 5.2: A scenario that all implementations in the refined contract are vacuous implementations since they form an empty set when intersecting with the original contract assumption.

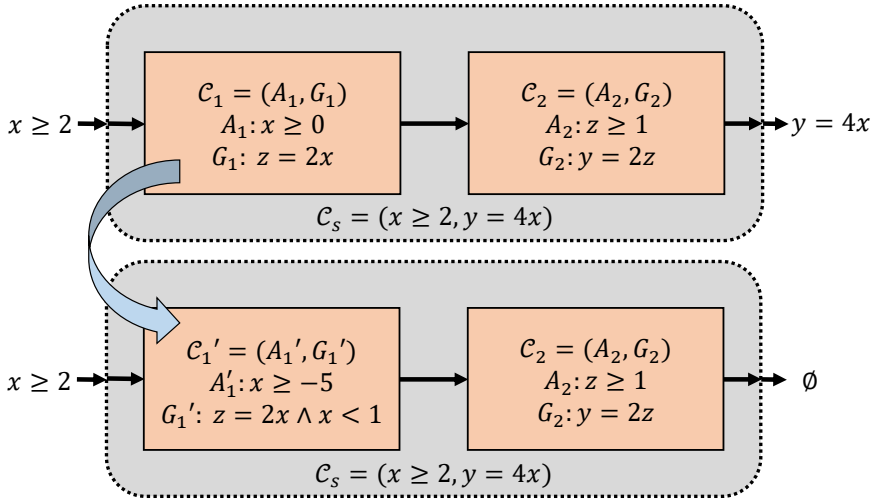


Figure 5.3: A motivating example that shows the vacuous implementation problem in contract refinement. All implementations based on the refined composition $\mathcal{C}_1 \parallel \mathcal{C}_2$ are vacuous implementations for $\mathcal{C}_s$.

disjoint, as illustrated in Figure 5.2. All implementations M' of the contract $\mathcal{C}'$ are vacuous implementations since $M' \cap A \subseteq (G' \cup \overline{A'}) \cap A = \emptyset$.

Figure 5.3 shows an example that the refinement of contracts results in vacuous implementations. Let the system contract be $\mathcal{C}_s = (A_s, G_s) = (x \geq 2, y = 4x)$, and two contracts be $\mathcal{C}_1 = (A_1, G_1) = (x \geq 0, z = 2x)$ and $\mathcal{C}_2 = (A_2, G_2) = (z \geq 1, y = 2z)$ as its subsystems whose composition refines the system contract. All contracts are defined on the system ports $\mathcal{P}_s = \{x, y, z\}$. The two subsystem contracts are then sent to different suppliers for independent development. If the supplier for $\mathcal{C}_1$ refines the contract as

$\mathcal{C}_1' = (A_1', G_1') = (x \geq -5, z = 2x \wedge x < 1)$ during the design process, the composition of $\mathcal{C}_1'$ and $\mathcal{C}_2$ remains a refinement for the system-level contracts. However, all implementations of the composition are vacuous implementations for $\mathcal{C}_s$, as shown in the following derivation:

$$\begin{aligned}(M_1' \cap M_2) \cap A_s &\subseteq (G_1' \cup \overline{A_1'}) \cap (G_2 \cup \overline{A_2}) \cap A_s \\ &\subseteq (G_1' \cup \overline{A_1'}) \cap A_s \\ &\subseteq ((z = 2x \wedge x < 1) \vee (x < -5)) \wedge (x \geq 2) \\ &= \emptyset,\end{aligned}$$

where M_1' is any implementation for $\mathcal{C}_1'$ and M_2 denotes any implementation for $\mathcal{C}_2$.

The example shows that the refinement can result in the vacuous implementation problem during independent design. Therefore, this work aims to restrict the refinement to guarantee strict implementations, and the vacuous implementation problem can avoid the problem in independent development as long as all suppliers follow the restriction. Our contributions are the following:

- We identify the vacuous implementation problem as an obstacle to the independent design paradigm. To the best of our knowledge, this is the first work that discovers and addresses this problem of the application of the contract-based design methodology to independent design.
- We introduce *replaceability*, a binary relation, as a sufficient condition to prevent the vacuous implementation problem. A refinement of a contract that follows *replaceability* is guaranteed to contain implementations compatible with the contract.
- We then introduce *strong replaceability*, a transitive binary relation, as a restriction on refinement to prevent the vacuous implementation problem in successive refinement steps and enable the independent design. Strong replaceability relieves the need for system contracts to enforce the requirement of replaceability. Once each refinement step follows *strong replaceability*, the resulting contract must follow replaceability with the system contract.
- We introduce the concept of *receptiveness* as a property of assume-guarantee contracts. We show that receptiveness is sufficient to ensure strong replaceability, and the independent design paradigm is permitted on receptive contracts for refinement and cascade composition.

The remainder of the section is organized as follows: Section 5.2.1 introduces the concept of assumption port sets and non-assumption port sets. Section 5.2.2 presents the related work. Section 5.2.3 formulates the contract replaceability requirement. Section 5.2.4 proposes the notion of contract receptiveness. Section 5.2.5 and Section 5.2.6 show that contract receptiveness ensures strong replaceability in refinement and cascade composition, respectively. Finally, Section 5.2.8 concludes the section.

5.2.1 Assumption Port Set

For a given contract $\mathcal{C} = (A, G)$ we define the non-assumption ports $\mathcal{P}_G$ and the assumption port set $\mathcal{P}_A$. The non-assumption port set is a subset of $\mathcal{P}_\mathcal{C}$ that is insensitive to the assumption set, defined formally as follows:

$$\mathcal{P}_G = \left\{ p \in \mathcal{P}_\mathcal{C} \;\middle|\; \begin{array}{l} \forall e \in \mathcal{B}_\mathcal{P} \\ (\pi_{\mathcal{P}_\mathcal{C}}(\pi_{\mathcal{P}_\mathcal{C}/\{p\}}(e)) \subseteq A) \vee \\ (\pi_{\mathcal{P}_\mathcal{C}}(\pi_{\mathcal{P}_\mathcal{C}/\{p\}}(e)) \subseteq \overline{A}) \end{array} \right\}.$$

The intuition of the definition is that the value of the non-assumption port does not affect the satisfaction of the assumption for all behaviors. The set $\pi_{\mathcal{P}_\mathcal{C}}(\pi_{\mathcal{P}_\mathcal{C}/\{p\}}(e))$ contains all behaviors that have the same value as e for all ports except for v.

The assumption port set $\mathcal{P}_A$ is defined as $\mathcal{P}_A = \mathcal{P}_\mathcal{C} \setminus \mathcal{P}_G$, the set difference of $\mathcal{P}_\mathcal{C}$ and $\mathcal{P}_G$.

For example, considering the contract $\mathcal{C} = (A, G) = (x \geq 0 \wedge y \geq 0, z = x + y), \mathcal{P}_\mathcal{C} = \{x, y, z\}$, the non-assumption port set $\mathcal{P}_G$ is $\{z\}$ and the assumption port $\mathcal{P}_A$ is $\{x, y\}$.

5.2.2 Related Work

This section focuses on the problem in the application of refinement in contract-based design. Many works have proposed algorithms for verifying and generating refinement of contracts. Cimatti *et al.* [34, 35] proposed the property-based proof systems to check whether a system is refined by the submodule contracts. The algorithm tests whether the guarantees generated by all submodules satisfy the top-level guarantees, and any top-level environments operating with all submodules create an environment for each submodule. Le *et al.* [96] proposed a similar paradigm more generically by defining a set of metatheoretical operators which allows the proof strategy to apply in different contract frameworks. Iannopollo *et al.* [76] adopted a hierarchical verification strategy and proposed a library-based contract refinement checking algorithm. The algorithm utilizes pre-checked refinement relations in the library to accelerate the verification. Iannopollo *et al.* [71, 73] also proposed a counter-example guided inductive synthesis-based constrained synthesis flow to synthesize contracts from a library of components or contracts specified using linear temporal logic. Their subsequent work [75] improves the synthesis efficiency by hierarchically decomposing the contracts into smaller contracts. These works are not aware of the potential vacuous implementation problem in the refinement process, the key enabler of the independent design paradigm. To the best of our knowledge, this is the first work that formally defines the requirement for independent design using contracts and introduces constraints to address the problem. As a result, this work complements the algorithms and tools by identifying the requirements for ensuring independent design. By enforcing the requirements, independent design can be ensured using the contract-based design without worrying about the vacuous implementation problem.

Receptiveness is the foundation for our proposed receptive contracts. The concept of receptiveness, which originates from the implementation point of view, was first proposed in [20], where receptiveness is defined over behaviors as any values for the specified variables

corresponding to some behaviors restricted by an assertion. Then the consistency of a contract is defined as the guarantee being **u**-receptive, where **u** stands for the uncontrolled variables in the variable set. The same notion for receptiveness was also mentioned in the later works [17, 34, 43, 151], while the contract consistency was defined differently in [17, 81] as contracts containing nonempty implementations set.

However, these works do not show the relation of receptiveness to the ability of independent design, as they intend to ensure receptive implementations and semantically separate the responsibilities of the assumption and guarantees instead of independent design. Their definition based on predefined partitioning of variables also limits the application of contracts as it cannot apply to components without rigorous input-output ports such as ones with bidirectional ports. Furthermore, being **u**-receptive requires the guarantees to include behaviors rejected by its assumptions, and thus the guarantees have larger behavior set sizes and contain redundant information. Therefore, taking the notion of receptiveness for behaviors as the foundation, our work defines receptiveness for contracts which does not contain redundant information by requiring receptiveness only for the behaviors accepted by assumptions. We show that our proposed receptive contracts ensure independent design and it does not rely on predefined partitioning of controlled and uncontrolled variables.

5.2.3 Contract Replaceability: Requirement for Independent Design

We define contract replaceability as the requirement to guarantee strict implementation:

Definition 5.1. *Let $\mathcal{C}_1 = (A_1, G_1)$ and $\mathcal{C}_2 = (A_2, G_2)$ be saturated contracts that satisfy $\mathcal{C}_1 \preceq \mathcal{C}_2$ and share the same port set $\mathcal{P}_\mathcal{C}$ and assumption port set $\mathcal{P}_A$. We say that $\mathcal{C}_1$ is replaceable by $\mathcal{C}_2$, or that $\mathcal{C}_2$ replaces $\mathcal{C}_1$, if the following condition is satisfied:*

$$\exists e \in \pi_{\mathcal{P}_A}(A_1), \pi_{\mathcal{P}_\mathcal{C}}(e) \cap G_2 \neq \emptyset,$$

or, equivalently, $A_1 \cap G_2 \neq \emptyset$.

Contract replaceability requires a projected behavior e in the assumption set A_1 such that the intersection of the guarantee set and the behavior projected back to the entire port set $\mathcal{P}_\mathcal{C}$ is not an empty set. As a result, a behavior with the assignments of the assumption ports satisfying A_1, the targeted environment, can be found in G_2, the refined guarantee. A binary relation called the contract replaceability relation is defined as the set containing all contract pairs $(\mathcal{C}_1, \mathcal{C}_2)$ such that $\mathcal{C}_1$ replaces $\mathcal{C}_2$.

Contract replaceability ensures that the strict implementations for the original contracts can be found using the refined contract, summarized in Theorem 5.1:

Theorem 5.1. *Let $\mathcal{C}_1 = (A_1, G_1)$ and $\mathcal{C}_2 = (A_2, G_2)$ be saturated contracts over the same port set $\mathcal{P}_\mathcal{C}$ and assumption port set $\mathcal{P}_A$ such that $\mathcal{C}_2$ refines $\mathcal{C}_1$ and $\mathcal{C}_2$ replaces $\mathcal{C}_1$. Any implementation M_2 of $\mathcal{C}_2$ such that $M_2 \supseteq (G_2 \cap A_2)$ is a strict implementation for $\mathcal{C}_1$.*

Proof. We prove Theorem 5.1 by showing that $M_2 \cap A_1$ is not an empty set:

$$M_2 \cap A_1 \supseteq (G_2 \cap A_2) \cap A_1 = G_2 \cap A_1 \neq \emptyset,$$

where the equality is by the definition of refinement that $A_1 \subseteq A_2$, and the inequality is by the definition of contract replaceability. Therefore, $M_2 \cap A_1$ is a superset of a non-empty set, which means $M_2 \cap A_1$ is not an empty set and thus a strict implementation for $\mathcal{C}_1$. □

As a result, once the system contract is replaceable by the refined contract, we can find a strict implementation for the system contract using the refined contract.

However, we need the assumption set from the system contract to ensure contract replaceability. In independent design, the supplier does not obtain the system contract but relies on a refined contract. Intuitively, we can require that the supplier guarantees contract replaceability for the refined contract instead of the system contract. Unfortunately, the contract replaceability relation is not transitive. A contract replacing the refined contract is not guaranteed to replace the system contract, as shown in the following example:

Example 5.1. *Consider the following three contracts $\mathcal{C}_1, \mathcal{C}_2$, and $\mathcal{C}_3$, where $\mathcal{C}_1 \succeq \mathcal{C}_2 \succeq \mathcal{C}_3$:*

$$\begin{aligned}
\mathcal{C}_1 &= (A_1, G_1) = (x \geq 0, y = 2x) \\
\mathcal{C}_2 &= (A_2, G_2) = (x \geq -2, (y = 2x \wedge x \leq 4) \vee (x < -2)) \\
\mathcal{C}_3 &= (A_3, G_3) = (x \geq -4, (y = 2x \wedge x \leq -1) \vee (x < -4)).
\end{aligned}$$

We can see that $\mathcal{C}_2$ replaces $\mathcal{C}_1$, and that $\mathcal{C}_3$ replaces $\mathcal{C}_2$. However, $\mathcal{C}_3$ does not replace $\mathcal{C}_1$ as $A_1 \cap G_3 = (x \geq 0) \wedge ((y = 2x \wedge x \leq -1) \vee (x < -4)) = \emptyset$ ■

To address the problem, a transitive relation that guarantees strict implementation is required Thus, we propose *strong replaceability*:

Definition 5.2. *Let $\mathcal{C}_1 = (A_1, G_1)$ and $\mathcal{C}_2 = (A_2, G_2)$ be saturated contracts that satisfy $\mathcal{C}_1 \succeq \mathcal{C}_2$ and share the same port set $\mathcal{P}_\mathcal{C}$ and assumption port set $\mathcal{P}_A$. We say that $\mathcal{C}_1$ is strongly replaceable by $\mathcal{C}_2$, or $\mathcal{C}_2$ strongly replaces $\mathcal{C}_1$, if the following condition is satisfied:*

$$\forall e \in \pi_{\mathcal{P}_{A_1}}(A_1), \pi_{\mathcal{P}_{\mathcal{C}_1}}(e) \cap G_2 \neq \emptyset.$$

Strong replaceability requires that for all projected behaviors e in the assumption set A_1, the intersection of the guarantee set and the behavior projected back to the entire port set $\mathcal{P}_{\mathcal{C}_1}$ is not an empty set. As a result, for each assignment of the assumption ports satisfying A_1, we can always find a satisfying behavior in G_2. A binary relation called the strong replaceability relation is defined as the set containing all contract pairs $(\mathcal{C}_1, \mathcal{C}_2)$ such that $\mathcal{C}_1$ strongly replaces $\mathcal{C}_2$. The difference between replaceability and strong replaceability is the quantification of the projected behavior.

We can show that the strong replaceability relation is transitive:

Proposition 5.1. *Let $\mathcal{C}_1 = (A_1, G_1)$, $\mathcal{C}_2 = (A_2, G_2)$, and $\mathcal{C}_3 = (A_3, G_3)$ be saturated contracts over the same port set $\mathcal{P}_\mathcal{C}$ and assumption port set $\mathcal{P}_A$ such that $\mathcal{C}_1 \succeq \mathcal{C}_2 \succeq \mathcal{C}_3$. If $\mathcal{C}_2$ strongly replaces $\mathcal{C}_1$ and $\mathcal{C}_3$ strongly replaces $\mathcal{C}_2$, then $\mathcal{C}_3$ strongly replaces $\mathcal{C}_1$.*

Proof. Since $\mathcal{C}_3$ strongly replaces $\mathcal{C}_2$, by the definition of strong replaceability:

$$\forall e_2 \in \pi_{\mathcal{P}_A}(A_2), \pi_{\mathcal{P}_\mathcal{C}}(e_2) \cap G_3 \neq \emptyset. \tag{5.1}$$

And $A_1 \subseteq A_2$ by the definition of contract refinement, so their projected behavior sets also hold the subset relation: $\pi_{\mathcal{P}_A}(A_1) \subseteq \pi_{\mathcal{P}_A}(A_2)$.

Therefore, $\forall e_1 \in \pi_{\mathcal{P}_A}(A_1), e_1 \in \pi_{\mathcal{P}_A}(A_2)$, and thus e_1 satisfies the qualification for (5.1): $\forall e_1 \in \pi_{\mathcal{P}_A}(A_1), \pi_{\mathcal{P}_\mathcal{C}}(e_1) \cap G_3 \neq \emptyset$. By the definition of strong replaceability, $\mathcal{C}_3$ strongly replaces $\mathcal{C}_1$. □

Combining Theorem 5.1 and Proposition 5.1, we conclude that strong replaceability guarantees strict implementations during independent design in Theorem 5.2

Theorem 5.2. *Let $\mathcal{C}_1 = (A_1, G_1)$, $\mathcal{C}_2 = (A_2, G_2)$, $\mathcal{C}_3 = (A_3, G_3)$, $\ldots, \mathcal{C}_n = (A_n, G_n)$ be saturated contracts over the same port set $\mathcal{P}_\mathcal{C}$ and assumption port set $\mathcal{P}_A$ such that $\mathcal{C}_i \succeq \mathcal{C}_{i+1}$ for $i = 1 \ldots n-1$. If $\mathcal{C}_{i+1}$ strongly replaces $\mathcal{C}_i$ for $i = 1 \ldots n-1$, then any implementation M_n such that $M_n \supseteq A_n \cap G_n$ strictly implements $\mathcal{C}_1$.*

Therefore, we propose strong replaceability as the restriction for suppliers to perform contract refinement. As long as all the suppliers follow the restriction to ensure strong replaceability, strict implementations for the system contracts can be found by $A_n \cap G_n$.

5.2.4 Receptive Contracts

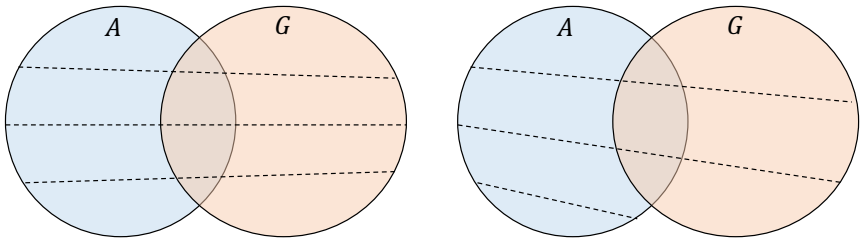
We have formulated strong replaceability as a restriction to ensure strict implementations in independent design. However, the problem that the conventional operations in assume-guarantee contracts cannot ensure strict implementations is worth exploring. In this section, we propose contract receptiveness as a constraint for assume-guarantee contracts so that any operations in independent design under the constraint ensure strict implementations. We will show that the receptive contract guarantees strong replaceability for refinement in Section 5.2.5 and cascade composition in Section 5.2.6.

Contract receptiveness is defined as follows:

Definition 5.3. *A receptive contract is a contract $\mathcal{C} = (A, G)$ satisfying the following condition:*

$$\forall e \in \pi_{\mathcal{P}_A}(A), \pi_{\mathcal{P}_\mathcal{C}}(e) \cap G \neq \emptyset.$$

A receptive contract requires that every assignment to the assumption port set allowed by the assumption set corresponds to at least a behavior in the guarantee set. Figure 5.4



(a) A receptive contract (b) A non-receptive contract

Figure 5.4: Illustrations of a receptive contract and a non-receptive contract. (a) A receptive contract as all its areas separated by the dashed lines intersect with the guarantee set. (b) A non-receptive contract as the area at the bottom of A does not intersect with the guarantee set.

illustrates the concept of the receptive contract. The areas between the dashed line represent all the assumption set assignments, $\pi_{\mathcal{P}_A}(A)$ Each area in the receptive contract, as shown in Figure 5.4 (a), must contain a behavior in G, while some areas in a non-receptive contract, as shown in Figure 5.4 (b), do not contain any behavior in G.

Example 5.2. *The contract $\mathcal{C}_1$ in Example 5.1 is a receptive contract while the contracts $\mathcal{C}_2$ and $\mathcal{C}_3$ are not receptive contracts. To check the receptiveness of $\mathcal{C}_1$, we first find the assumption set assignments $\pi_{\mathcal{P}_{A_1}}(A_1) = \{x \mid x \geq 0\}$. For all assignments $x \geq 0$, we can find a behavior $(x, y) = (x, 2x)$ that is in G_1 and $\pi_{\mathcal{P}_{\mathcal{C}_1}}(x)$. Therefore, $\mathcal{C}_1$ is a receptive contract.*

Then we check the receptiveness of the contracts $\mathcal{C}_2$ and $\mathcal{C}_3$ in Example 5.1, The assignments of the assumption port allowed by $\mathcal{C}_2$ is $\{x \mid x \geq -2\}$. However, as the guarantee set requires $x \leq 4$, any behavior with assignments of the assumption port being $x > 4$ is not in the guarantee set. Similarly, for $\mathcal{C}_3$, the guarantee set requires $x \leq -1$, and thus any behavior with assignments of the assumption port being $x > -1$ is not in the guarantee set. Therefore, the two contracts are not receptive. ■

5.2.5 Refinement with Receptive Contracts

In this section, we show that the proposed receptive contracts guarantee strong replaceability for refinement during independent design and allow the suppliers to discover design faults in the specifications.

Theorem 5.3 states that receptive contracts guarantee strong replaceability in the refinement operation:

Theorem 5.3. *Let $\mathcal{C}_1 = (A_1, G_1)$ and $\mathcal{C}_2 = (A_2, G_2)$ be saturated contracts over the same port set $\mathcal{P}_\mathcal{C}$ and assumption port set $\mathcal{P}_A$ such that $\mathcal{C}_1 \succeq \mathcal{C}_2$. If $\mathcal{C}_2$ is a receptive contract, then $\mathcal{C}_2$ strongly replaces $\mathcal{C}_1$.*

Proof. We prove Theorem 5.3 by converting the condition for receptiveness to the condition for strong replaceability:

As $\mathcal{C}_2$ is receptive, using the definition for the receptive contract we get:

$$\forall e_1 \in \pi_{\mathcal{P}_A}(A_2), G_2 \cap \pi_{\mathcal{P}_C}(e_1) \neq \emptyset.$$

By the definition of contract refinement, $A_1 \subseteq A_2$, so $\pi_{\mathcal{P}_A}(A_1) \subseteq \pi_{\mathcal{P}_A}(A_2)$. Therefore, $\forall e_2 \in \pi_{\mathcal{P}_A}(A_1), e_2 \in \pi_{\mathcal{P}_A}(A_2)$. Combining the above results, we get

$$\forall e_2 \in \pi_{\mathcal{P}_A}(A_1), G_2 \cap \pi_{\mathcal{P}_C}(e_2) \neq \emptyset.$$

Therefore, $\mathcal{C}_2$ strongly replaces $\mathcal{C}_1$ by the definition of strong replaceability. □

Receptive contracts guarantee strong replaceability not only for the abstract contracts before refinement but also for the system contract. To see this, we first show that a receptive refined contract implies that its abstract contract is also receptive:

Proposition 5.2. *Let $\mathcal{C}_1 = (A_1, G_1)$ and $\mathcal{C}_2 = (A_2, G_2)$ be saturated contracts over the same port set $\mathcal{P}_C$ and assumption port set $\mathcal{P}_A$ such that $\mathcal{C}_1 \preceq \mathcal{C}_2$. If $\mathcal{C}_2$ is a receptive contract, then $\mathcal{C}_1$ is also a receptive contract.*

Proof. During the proof in Theorem 5.3, we have derived the following:

$$\forall e_2 \in \pi_{\mathcal{P}_A}(A_1), G_2 \cap \pi_{\mathcal{P}_C}(e_2) \neq \emptyset.$$

By the definition of contract refinement, $G_1 \supseteq G_2$:

$$\forall e_2 \in \pi_{\mathcal{P}_A}(A_1), G_1 \cap \pi_{\mathcal{P}_C}(e_2) \supseteq G_2 \cap \pi_{\mathcal{P}_C}(e_2) \neq \emptyset.$$

Therefore, $\mathcal{C}_1$ is a receptive contract by Definition 5.3. □

Strong replaceability for the system contracts, summarized in Theorem 5.4, can thus be derived by combining Proposition 5.2 and Theorem 5.3:

Theorem 5.4. *Let $\mathcal{C}_1 = (A_1, G_1)$, $\mathcal{C}_2 = (A_2, G_2)$, $\mathcal{C}_3 = (A_3, G_3)$, $\ldots, \mathcal{C}_n = (A_n, G_n)$ be saturated contracts over the same port set $\mathcal{P}_C$ and assumption port set $\mathcal{P}_A$ such that $\mathcal{C}_i \preceq \mathcal{C}_{i+1}$ for $i = 1 \ldots n-1$. If $\mathcal{C}_n$ is a receptive contract, then $\mathcal{C}_n$ strongly replaces $\mathcal{C}_1$.*

Proof. We prove Theorem 5.4 by induction. When $n = 2$, the statement holds by Theorem 5.3. Assume that the statement holds for $n = k$. When $n = k + 1$, $\mathcal{C}_{k+1}$ strongly replaces $\mathcal{C}_2$ by the assumption. $\mathcal{C}_2$ is a receptive contract as $\mathcal{C}_{k+1}$ is a receptive contract by Proposition 5.2. Applying Theorem 5.3 on $\mathcal{C}_2$ and $\mathcal{C}_1$, $\mathcal{C}_2$ strongly replaces $\mathcal{C}_1$. Therefore, by the transitivity of strong replaceability in Proposition 5.1, $\mathcal{C}_{k+1}$ strongly replaces $\mathcal{C}_1$. By mathematical induction, Theorem 5.4 holds for any $n \geq 2$. □

Furthermore, Theorem 5.4 implies that the suppliers can discover faults in the specification. We can impose receptiveness as a constraint on the assume-guarantee contract. The suppliers can rest assured that strong replaceability holds as long as the received abstract contract is receptive. In contrast, if the suppliers receive a non-receptive abstract contract, some faults must have occurred before the abstract contract was generated. Accordingly, the supplier can alert the specification provider to check for faults during the design process.

5.2.6 Cascade Composition with Receptive Contracts

We have presented requirements for independent design under the refinement operations. However, in contract-based design, an abstract contract can be decomposed into several contracts whose composition refines the abstract contract. The decomposition of a contract is analogous to decomposing a system into several subsystems. Each subsystem follows the decomposed contracts. If a supplier receives one of the subsystem contracts and refines the subsystem contract, the supplier cannot check the strong replaceability of the composition without the other subsystem contracts. In this section, we discuss strong replaceability in composition using receptive contracts.

A composition is either a cascade composition or a feedback composition, depending on the topology of the subsystems. A cascade composition has subsystem order such that the assumption port set of each subsystem only connects to the ports from the assumption port set of the environment or the ports set from the preceding subsystems. We will use the subscript s to denote the system contract and numbers as subscripts to denote the order for the subsystem in a cascade composition. For example, let a system $\mathcal{C}_s$ be a cascade composition of two subsystem contracts $\mathcal{C}_1 \parallel \mathcal{C}_2$. Then $\mathcal{C}_1$ precedes $\mathcal{C}_2$, and thus by the definition of cascade composition, $\mathcal{P}_{A_1}$ must be a subset of $\mathcal{P}_{A_s}$. A feedback composition is any composition that is not a cascade composition, meaning that subsystem order cannot be defined.

In this section, we discuss the following problem: Let the system contract be $\mathcal{C}_s = (A_s, G_s)$, Let $\mathcal{C}_1 = (A_1, G_1)$ and $\mathcal{C}_2 = (A_2, G_2)$ be saturated receptive contracts in a cascade composition such that $\mathcal{C}_s \succeq \mathcal{C}_1 \parallel \mathcal{C}_2$. Let $\mathcal{C}'_1 = (A'_1, G'_1)$ and $\mathcal{C}'_2 = (A'_2, G'_2)$ be saturated receptive contracts such that $\mathcal{C}_1 \succeq \mathcal{C}'_1$ and $\mathcal{C}_2 \succeq \mathcal{C}'_2$. All the behavior sets are defined in the port set $\mathcal{P}_{\mathcal{C}_s}$. We will show that the composition $\mathcal{C}'_1 \parallel \mathcal{C}'_2$ strongly replaces the system contract $\mathcal{C}_s = (A_s, G_s)$. For the feedback composition, we will present an example showing that the composition of the refined receptive contracts does not ensure strong replaceability. More constraints are thus required for general composition. We leave these additional constraints for feedback composition as future work.

We develop two lemmas to show strong replaceability of the cascade composition by receptive contracts. The first lemma, summarized in Lemma 5.1, states that any assignments to the assumption port set of the system contract must satisfy the assumption of the first contracts.

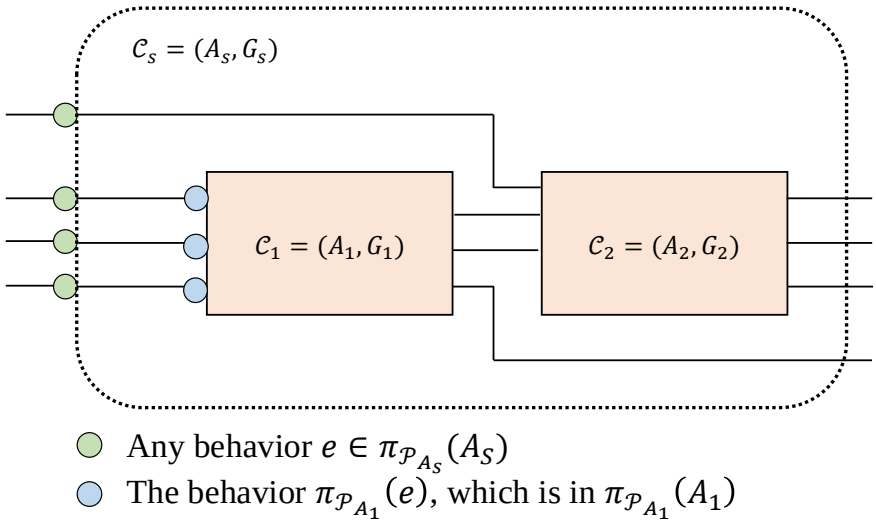


Figure 5.5: Visualization of Lemma 5.1. Any behavior from the targeted assumption satisfies the assumption of $\mathcal{C}_1$.

Lemma 5.1. *Let $\mathcal{C}_s = (A_s, G_s)$, $\mathcal{C}_1 = (A_1, G_1)$, and $\mathcal{C}_2 = (A_2, G_2)$ be saturated receptive contracts such that $\mathcal{C}_s \succeq \mathcal{C}_1 \parallel \mathcal{C}_2$, then $\forall e \in \pi_{\mathcal{P}_{A_s}}(A_s), \pi_{\mathcal{P}_{A_1}}(e) \in \pi_{\mathcal{P}_{A_1}}(A_1)$.*

Figure 5.5 illustrates the concept in Lemma 5.1. The lemma is proved by contradiction: if $\pi_{\mathcal{P}_{A_1}}(e) \notin \pi_{\mathcal{P}_{A_1}}(A_1)$, then $e \notin \pi_{\mathcal{P}_{A_s}}(A_s)$.

Proof. Assume that e is a counterexample of Lemma 5.1 such that $e \in \pi_{\mathcal{P}_{A_s}}(A_s)$ and $\pi_{\mathcal{P}_{A_1}}(e) \notin \pi_{\mathcal{P}_{A_1}}(A_1)$. We want show that $e \notin \pi_{\mathcal{P}_{A_s}}(A_s)$, and thus the assumption leads to a contradiction.

First, we show that $\pi_{\mathcal{P}_{\mathcal{C}_s}}(e) \subseteq \overline{A_1}$ and $\pi_{\mathcal{P}_{\mathcal{C}_s}}(e) \subseteq G_1$. Since $\pi_{\mathcal{P}_{A_1}}(e) \notin \pi_{\mathcal{P}_{A_1}}(A_1)$, we can project the two sides back to $\mathcal{P}_{\mathcal{C}_s}$:

$$\begin{aligned} & \pi_{\mathcal{P}_{A_1}}(e) \notin \pi_{\mathcal{P}_{A_1}}(A_1) \\ \implies & \pi_{\mathcal{P}_{\mathcal{C}_s}}(\pi_{\mathcal{P}_{A_1}}(e)) \subseteq \overline{\pi_{\mathcal{P}_{\mathcal{C}_s}}(\pi_{\mathcal{P}_{A_1}}(A_1))} \\ \implies & \pi_{\mathcal{P}_{\mathcal{C}_s}}(e) \subseteq \overline{\pi_{\mathcal{P}_{\mathcal{C}_s}}(A_1)} \\ \implies & \pi_{\mathcal{P}_{\mathcal{C}_s}}(e) \subseteq \overline{A_1} \subseteq G_1 \cup \overline{A_1} \subseteq G_1. \end{aligned} \tag{5.2}$$

Then we discuss whether e can satisfy the assumption set of the second contract in two cases. The first case is $\pi_{\mathcal{P}_{A_2}}(e) \subseteq \overline{\pi_{\mathcal{P}_{A_2}}(A_2)}$ and the second case is $\pi_{\mathcal{P}_{A_2}}(e) \nsubseteq \overline{\pi_{\mathcal{P}_{A_2}}(A_2)}$.

Case 1

$$\begin{aligned} & \pi_{\mathcal{P}_{A_2}}(e) \subseteq \overline{\pi_{\mathcal{P}_{A_2}}(A_2)} \\ \Longrightarrow\ & \pi_{\mathcal{P}_{\mathcal{C}_s}}(\pi_{\mathcal{P}_{A_2}}(e)) \subseteq \overline{\pi_{\mathcal{P}_{\mathcal{C}_s}}(\pi_{\mathcal{P}_{A_2}}(A_2))} \\ \Longrightarrow\ & \pi_{\mathcal{P}_{\mathcal{C}_s}}(e) \subseteq \overline{\pi_{\mathcal{P}_{\mathcal{C}_s}}(A_2)} \\ \Longrightarrow\ & \pi_{\mathcal{P}_{\mathcal{C}_s}}(e) \subseteq \overline{A_2} \subseteq G_2 \cup \overline{A_2} = G_2. \end{aligned} \tag{5.3}$$

Combining (5.2) and (5.3), we get:

$$\begin{aligned} \pi_{\mathcal{P}_{\mathcal{C}_s}}(e) &\subseteq (\overline{A_1} \cup \overline{A_2}) \cap (G_1 \cap G_2) \\ &\subseteq \overline{((A_1 \cap A_2) \cup \overline{(G_1 \cap G_2)})} \\ &\subseteq \overline{A_s}. \end{aligned}$$

Therefore, $\pi_{\mathcal{P}_{\mathcal{C}_s}}(e) \subseteq \overline{A_s}$, and thus $e \notin \pi_{\mathcal{P}_{A_s}}(A_s)$, which contradicts our assumption that e is a counterexample.

Case 2 When $\pi_{\mathcal{P}_{A_2}}(e) \nsubseteq \overline{\pi_{\mathcal{P}_{A_2}}(A_2)}$, $\pi_{\mathcal{P}_{A_2}}(e) \cap \pi_{\mathcal{P}_{A_2}}(A_2) \neq \emptyset$. We can find a behavior $e_2 \in \pi_{\mathcal{P}_{A_2}}(e) \cap \pi_{\mathcal{P}_{A_2}}(A_2)$. Since $\mathcal{C}_2$ is a receptive contract, we can find a behavior $e_3 \in \pi_{\mathcal{P}_{\mathcal{C}_2}}(G_2) \cap \pi_{\mathcal{P}_{\mathcal{C}_2}}(e_2)$. Considering the behavior $e_4 = \pi_{\mathcal{P}_{\mathcal{C}_s}}(e_3) \cap \pi_{\mathcal{P}_{\mathcal{C}_s}}(e)$, we can get $e_4 \in \pi_{\mathcal{P}_{\mathcal{C}_s}}(e)$ and $e_4 \in \pi_{\mathcal{P}_{\mathcal{C}_s}}(e_3)$. Also, $e_4 \in \pi_{\mathcal{P}_{\mathcal{C}_s}}(e)$ implies $e_4 \in \overline{A_1} \subseteq G_1$.

Therefore, we can get:

$$e_3 \in \pi_{\mathcal{P}_{\mathcal{C}_2}}(G_2) \implies \pi_{\mathcal{P}_{\mathcal{C}_s}}(e_3) \subseteq G_2 \implies e_4 \in G_2.$$

As a result, we can derive that e_4 is not a behavior in A_s:

$$\begin{aligned} e_4 &\in \overline{A_1} \cap (G_1 \cap G_2) \\ &\in \overline{((A_1 \cap A_2) \cup \overline{(G_1 \cap G_2)})} \\ &\in \overline{A_s}. \end{aligned}$$

Therefore, $e_4 \in \overline{A_s}$, and thus $e = \pi_{\mathcal{P}_{A_s}}(e_4) \notin \pi_{\mathcal{P}_{A_s}}(A_s)$, which contradicts the assumption that e is a counterexample.

As both cases lead to contradictions, Lemma 5.1 is thus proved. □

The other lemma, as shown in Lemma 5.2, states that the behaviors of the first contract satisfy the assumption of the second contract if the behaviors meet the assumption of the system contract.

Lemma 5.2. *Let $\mathcal{C}_s = (A_s, G_s)$, $\mathcal{C}_1 = (A_1, G_1)$, and $\mathcal{C}_2 = (A_2, G_2)$ be saturated receptive contracts such that $\mathcal{C}_s \succeq \mathcal{C}_1 \parallel \mathcal{C}_2$, then $\forall e_1 \in \pi_{\mathcal{P}_{A_s}}(A_s), \forall e_2 \in \pi_{\mathcal{P}_{\mathcal{C}_1}}(G_1) \cap \pi_{\mathcal{P}_{\mathcal{C}_1}}(e_1), \pi_{\mathcal{P}_{A_2}}(e_1) \cap \pi_{\mathcal{P}_{A_2}}(e_2) \in \pi_{\mathcal{P}_{A_2}}(A_2)$.*

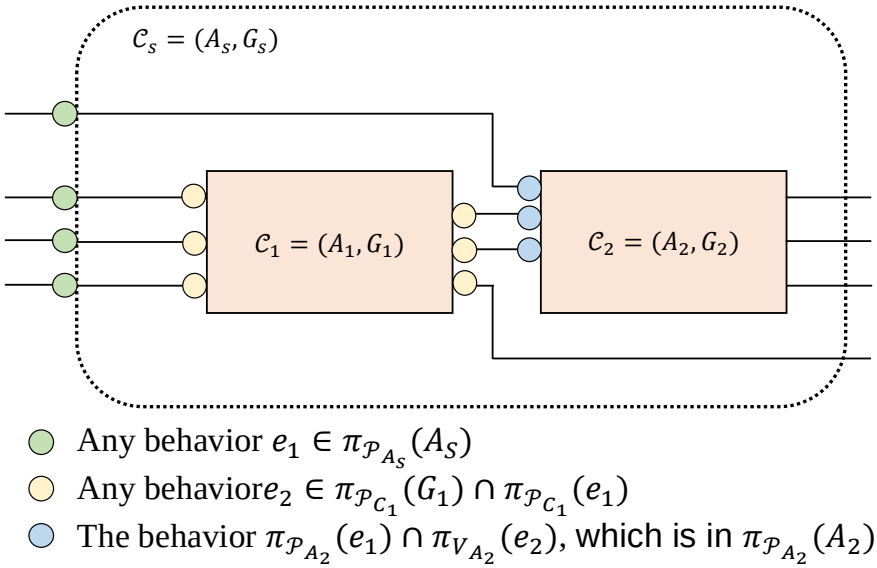


Figure 5.6: Visualization of Lemma 5.2. The combined behavior of any behavior from the targeted assumption and the corresponding behavior generated by $\mathcal{C}_1$ satisfies the assumption of $\mathcal{C}_2$.

Figure 5.6 illustrates the concept in Lemma 5.2. The projected assumption from the system contracts and all the corresponding behaviors generated by G_1 must be in the assumption set of the second contract.

The lemma is proved by contradiction that if $\pi_{\mathcal{P}_{A_1}}(e_1) \notin \pi_{\mathcal{P}_{A_1}}(A_1)$, then $e \notin \pi_{\mathcal{P}_{A_s}}(A_s)$:

Proof. Assume that $e \in \pi_{\mathcal{P}_{A_s}}(A_s)$ and that $e_2 \in \pi_{\mathcal{P}_{\mathcal{C}_1}}(G_1) \cap \pi_{\mathcal{P}_{\mathcal{C}_1}}(e_1)$ forms a counterexample such that $\pi_{\mathcal{P}_{A_2}}(e_1) \cap \pi_{\mathcal{P}_{A_2}}(e_2) \notin \pi_{\mathcal{P}_{A_2}}(A_2)$. Therefore, we can derive the following:

$$\begin{aligned} &\pi_{\mathcal{P}_{A_2}}(e_1) \cap \pi_{\mathcal{P}_{A_2}}(e_2) \notin \pi_{\mathcal{P}_{A_2}}(A_2) \\ \implies &\pi_{\mathcal{P}_{\mathcal{C}_s}}(e_1) \cap \pi_{\mathcal{P}_{\mathcal{C}_s}}(e_2) \subseteq \overline{A_2} \subseteq G_2 \end{aligned}$$

and

$$\begin{aligned} &e_2 \in \pi_{\mathcal{P}_{\mathcal{C}_1}}(G_1) \\ \implies &\pi_{\mathcal{P}_{\mathcal{C}_s}}(e_2) \subseteq G_1 \\ \implies &\pi_{\mathcal{P}_{\mathcal{C}_s}}(e_1) \cap \pi_{\mathcal{P}_{\mathcal{C}_s}}(e_2) \subseteq G_1. \end{aligned}$$

Similar to the proof for Lemma 5.1, we can show that $\pi_{\mathcal{P}_{\mathcal{C}_s}}(e_1) \cap \pi_{\mathcal{P}_{\mathcal{C}_s}}(e_2) \subseteq \overline{A_s}$:

$$\begin{aligned} \pi_{\mathcal{P}_{\mathcal{C}_s}}(e_1) \cap \pi_{\mathcal{P}_{\mathcal{C}_s}}(e_2) &\subseteq (\overline{A_2}) \cap (G_1 \cap G_2) \\ &\subseteq \overline{((A_1 \cap A_2) \cup \overline{(G_1 \cap G_2)})} \\ &\subseteq \overline{A_s}. \end{aligned}$$

Therefore, $\pi_{\mathcal{P}_{\mathcal{C}_s}}(e_1) \subseteq \overline{A_s}$, and thus $e_1 \notin \pi_{\mathcal{P}_{A_s}}(A_s)$ contradicts the assumption. □

With Lemmas 5.1 and 5.2, we conclude that any refinement to the receptive contracts ensures strong replaceability, as shown in Theorem 5.5:

Theorem 5.5. *Let $\mathcal{C}_s = (A_s, G_s)$, $\mathcal{C}_1 = (A_1, G_1)$, and $\mathcal{C}_2 = (A_2, G_2)$ be saturated receptive contracts such that $\mathcal{C}_s \succeq \mathcal{C}_1 \parallel \mathcal{C}_2$, and let $\mathcal{C}_1' = (A_1', G_1')$ and $\mathcal{C}_2' = (A_2', G_2')$ be saturated receptive contracts such that $\mathcal{C}_1 \succeq \mathcal{C}_1'$ and $\mathcal{C}_2 \succeq \mathcal{C}_2'$, then $\mathcal{C}_1' \parallel \mathcal{C}_2'$ strongly replaces $\mathcal{C}_s$.*

Proof. Using the proposition of contract refinement, $\mathcal{C}_s \succeq \mathcal{C}_1' \parallel \mathcal{C}_2'$. By Lemma 5.1, for every $e \in \pi_{\mathcal{P}_{A_s}}(A_s)$, $\pi_{\mathcal{P}_{A_1}}(e) \in \pi_{\mathcal{P}_{A_1}}(A_1')$. Since $\mathcal{C}_1'$ is a receptive contract, we can find e_2 such that $e_2 \in \pi_{\mathcal{P}_{\mathcal{C}_1}}(e) \cap \pi_{\mathcal{P}_{\mathcal{C}_1}}(G_1')$, and thus:

$$\pi_{\mathcal{P}_{\mathcal{C}_s}}(e_2) \in G_1'. \tag{5.4}$$

By Lemma 5.2, e_2 and e satisfies $\pi_{\mathcal{P}_{A_2}}(e) \cap \pi_{\mathcal{P}_{A_2}}(e_2) \in \pi_{\mathcal{P}_{A_2}}(A_2)$. Similarly, since $\mathcal{C}_2'$ is a receptive contract, we can find e_3 such that $e_3 \in \pi_{\mathcal{P}_{\mathcal{C}_2}}(\pi_{\mathcal{P}_{A_2}}(e) \cap \pi_{\mathcal{P}_{A_2}}(e_2)) \cap \pi_{\mathcal{P}_{\mathcal{C}_2}}(G_2')$, and thus:

$$\pi_{\mathcal{P}_{\mathcal{C}_s}}(e_3) \in G_2'. \tag{5.5}$$

Considering the behavior $\pi_{\mathcal{P}_{\mathcal{C}_s}}(e) \cap \pi_{\mathcal{P}_{\mathcal{C}_s}}(e_2) \cap \pi_{\mathcal{P}_{\mathcal{C}_s}}(e_3)$ and combining the results in (5.4) and (5.5), we get:

$$\begin{aligned}\pi_{\mathcal{P}_{\mathcal{C}_s}}(e) \cap \pi_{\mathcal{P}_{\mathcal{C}_s}}(e_2) \cap \pi_{\mathcal{P}_{\mathcal{C}_s}}(e_3) &\in G_1' \cap G_2' \\ &\subseteq G_s.\end{aligned}$$

As $\pi_{\mathcal{P}_{\mathcal{C}_s}}(e) \cap \pi_{\mathcal{P}_{\mathcal{C}_s}}(e_2) \cap \pi_{\mathcal{P}_{\mathcal{C}_s}}(e_3) \in \pi_{\mathcal{P}_{\mathcal{C}_s}}(e)$, the condition for strong replaceability is satisfied. Therefore, $\mathcal{C}_1' \parallel \mathcal{C}_2'$ strongly replaces $\mathcal{C}_s$. □

Finally, we show an example of a feedback composition using receptive contracts that contains only vacuous implementations after refinement:

Example 5.3. *Let $\mathcal{C}_s$ be the system contract, $\mathcal{C}_1$ and $\mathcal{C}_2$ be the subsystem contracts, and $\mathcal{C}_1'$ be the refined contract for $\mathcal{C}_1$:*

$$\begin{aligned}&\mathcal{C}_s = (True, y = \frac{x}{1-x}), \mathcal{P}_s = \{x, y\},\\ &\mathcal{C}_1 = (True, (y = b + 1) \vee (y = xb)), \mathcal{P}_1 = \{x, y, b\},\\ &\mathcal{C}_2 = (True, b = y + 1), \mathcal{P}_2 = \{y, b\},\\ &\mathcal{C}_1' = (True, y = b + 1), \mathcal{P}_1{}' = \{x, y, b\}.\end{aligned}$$

The compositions $\mathcal{C}_1' \parallel \mathcal{C}_2$ and $\mathcal{C}_1 \parallel \mathcal{C}_2$ both refine $\mathcal{C}_1$. But $\mathcal{C}_1' \parallel \mathcal{C}_2 = (True, \emptyset)$, and thus the only implementation is a vacuous implementation $M_1' = \emptyset$, even though the refined contract is a receptive contract. ■

Therefore, additional constraints are needed for feedback composition such that the strong replaceability of any composition is ensured. The constraints for the feedback composition will be explored in Section 5.3.

5.2.7 Discussion

In this section, we discuss the impacts of the discovery and proposed concept on the contract-based design process, and thus the need for the development of new algorithms and tools for supporting contract-based design.

5.2.7.1 Design Faults in Refinement

The vacuous implementation problem should be regarded as a type of design fault, which might be caused by the designer or problems in the automation tools to generate refined contracts not satisfying the replaceability relation. The replaceability relation is crucial for the refinement process to guarantee the compatibility of its subsystems and thus avoid vacuous implementations after system integration. Only verifying the refinement relation cannot capture this type of design fault. Therefore, existing contract-based design methodologies [151, 127] that propose using refinement in the design process, should include a stage for verifying the replaceability of the top-level specification. The transitive strong replaceability breaks down the problem of verifying the replaceability of the top-level specification into verifying the strong replaceability between each refinement step and thus can be applied in the independent design paradigm. If the design faults are not captured in this early stage of design, the vacuous implementation would result in huge costs and design time overhead.

5.2.7.2 Applying Receptive Contracts

In Section 5.2.4– 5.2.6, we have shown that receptive contracts guarantee strong replaceability in cascade composition and pure contract refinement. The theory indicates that using receptive contracts can further simplify the process of verifying the replaceability relation. As long as the system does not contain feedback composition, receptive subsystem contracts guarantee the replaceability of the refined systems to the top-level specifications. In many application fields, the specifications should be receptive by their definitions, such as controller design and sequential programs. The inputs and outputs are explicitly defined for every system in these fields. Therefore, verifying receptive contracts can serve two roles at the same time, one is verifying the design faults, and the other is maintaining the semantics of the components, as it is meaningless for a controller or a program method to have no outputs for any allowable inputs.

5.2.7.3 The Need for Development of New Algorithms and Tools

With the proposed theory, we suggest the development of new algorithms and tools to facilitate the contract-based design. Existing contract tools and algorithms [34, 76] do not include the functionality to verify the replaceability relation, and thus are unable to detect the design faults of vacuous implementation. The universal quantification in the strong replaceability and receptiveness is challenging for algorithm development as its decidability depends on the representations of contracts. For example, the Presburger arithmetic is

decidable while it becomes undecidable if multiplication is involved [115]. Therefore, research on tools and algorithms for different representations of contracts is required to prevent design faults and enable independent design using contracts.

5.2.8 Conclusion

We identified the vacuous implementation problem using assume-guarantee contracts under the independent design paradigm. We first explored the notion of contract replaceability. This notion was shown to be the requirement to ensure strict implementations, but it is not transitive, thus limiting its applicability in independent design. The stricter notion of strong replaceability also ensures strict implementations and is transitive, thus fitting the independent design paradigm. We then proposed the notion of contract receptiveness, which guarantees strong replaceability. Moreover, we showed that receptive contracts can be implemented independently and that the composition of their implementations will not be vacuous in the case of cascade composition. A supplier receiving a contract as the specification for implementation can check whether this contract is receptive. If so, the supplier knows in advance that it can proceed to develop an implementation and that this implementation will integrate correctly into the system integrator's design. Our areas of future work include finding constraints for feedback composition, developing tools to support independent design, and investigating the replaceability in different contract formalisms.

5.3 Ensuring Correct Decomposition of Assume-Guarantee Contracts in Feedback Composition

In contract-based design methodologies, the design process involves iterative decompositions of the contract into independent subsystem contracts until each design problem is manageable [17]. The integration of the implementations of these subsystem contracts constitutes the system implementation. This compositional approach offers the potential for reducing design complexity and handling system heterogeneity. Figure 5.7(a) illustrates the concept of decomposition as a refinement result, and each decomposed contract can be developed independently. By leveraging composition and refinement, designers can decompose system design problems into independent subsystem contracts, facilitating development while ensuring overall contract satisfaction.

However, as mentioned in the previous section, refinement only pessimistically ensures that the decomposition result never violates the requirement of the system contract; the subsystems producing empty behaviors under the targeted environment can vacuously meet the refinement relation. These subsystems do not meet the design goal as empty behaviors mean that they cannot operate in such an environment, suggesting that the decomposition that leads to such subsystems is undesired. Consequently, using refinement as the sole criterion to

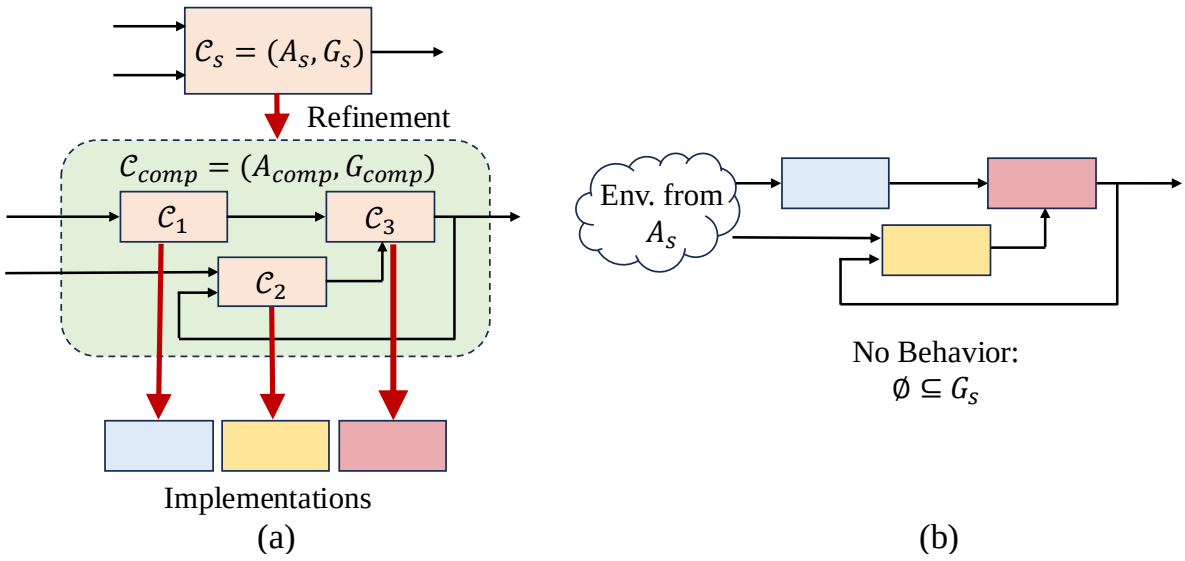


Figure 5.7: Illustration of the problematic decomposition and vacuous implementation (a) a decomposition that satisfies refinement relation with implementations for the subsystem contracts and (b) the overall implementation that may have zero behavior under the environment from A_s while does not violate refinement relation.

examine contract decomposition may result in a problematic decomposition not suitable for independent development. This problem is called *vacuous implementation problem*, where vacuous implementations are ones that have empty behaviors under the targeted environment [182]. The issue is due to the lack of guarantees regarding the existence of behaviors, as the framework only ensures the specifications are never violated. Figure 5.7(b) depicts this concept, the empty set can vacuously satisfy the refinement relation: $\emptyset \subseteq G_s$, making it a potential result after independent development. Consequently, contracts with empty behaviors under the targeted environment can satisfy the refinement relation, leading to vacuous implementations. The failure to meet the design goal compromises the benefits of independent development in contract-based design, as it does not ensure correct implementation throughout the process.

While the previous section introduced strong replaceability and receptiveness to address the vacuous implementation problem — where relying solely on refinement relationships leads to faulty decomposition — strong replaceability in feedback composition is not guaranteed. As a result, a gap in the contract theory is the conditions necessary to ensure strong replaceability within feedback composition, and how these conditions can be verified. Two main drawbacks result from the gap. First, the designers of each subsystem involved in feedback composition must communicate all design decisions to the subsystem to ensure strong replaceability between the composition of the subsystems and the system contract. As the subsystem design problems are no longer independent, the potential benefits from complex-

ity reduction are compromised. Second, the entire system contract must be disclosed to all levels of designers to prevent vacuous implementations, increasing the risk of trade secret leaks as more designers have access to the system contract.

To overcome these drawbacks, this work aims to guarantee correct contract decomposition by ensuring strong replaceability in feedback composition. Our contributions are the following:

- We formulate the verification problem to ensure correct contract decomposition by establishing strong replaceability for any individual contract refinement within feedback composition.
- We define the behaviors in feedback composition as *fixed obligations* and introduce *fixed obligation graphs* to convert the verification problem into a graph-based problem. This aids in visualizing the relation between fixed obligations and facilitates proofs by graph theory.
- We identify the necessary and sufficient conditions required to address the verification problem utilizing the fixed obligation graphs. By incorporating existing receptiveness and strong replaceability, we fully resolve the vacuous implementation issues in assume-guarantee contracts, ensuring the benefits of independent development.
- We propose verification algorithms based on the developed conditions. These algorithms can serve as a foundation for automatic verification for contract-based design.

The remainder of this section is organized as follows: Section 5.3.1 provides definitions for feedback composition. Section 5.3.2 formulates the verification problem for strong replaceability. Section 5.3.3 defines the fixed obligations and introduces the fixed obligation graphs. Section 5.3.4 presents the necessary and sufficient conditions to ensure strong replaceability. Section 5.3.5 proposes verification algorithms based on the identified conditions. Finally, Section 5.3.6 concludes the section.

5.3.1 Feedback Composition and Port Partition

A composition is a feedback one when no subsystem orders can be established so that each subsystem's input ports only connect to the environment or ports from preceding subsystems. Fig. 5.8(a) shows an example system s consisting of two subsystems s_1 and s_2.

In this section, we assume that connected ports are fully compatible and can, for simplicity, be treated as identical, eliminating the need for lengthy expressions about port value equivalence. This simplification does not affect the generality of the discussion, as port compatibility can be encoded in the contract assumptions.

Based on the relation of the ports, we can partition the ports in feedback composition as follows:

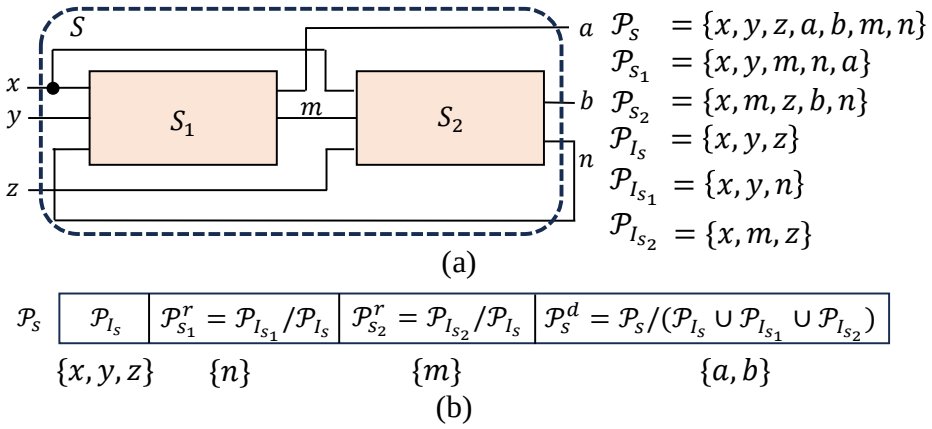


Figure 5.8: An example of (a) a feedback composition and (b) its port partition.

- *System Inputs* $\mathcal{P}_{I_s}$: The input ports of the system, which can also be derived from $(\mathcal{P}_{I_{s_1}}/\mathcal{P}_{O_{s_2}}) \cup (\mathcal{P}_{I_{s_2}}/\mathcal{P}_{O_{s_1}})$.
- *Related Inputs* $\mathcal{P}^r_{s_1}$ and $\mathcal{P}^r_{s_2}$: The input ports of a subsystem that rely on the output of the other subsystem.
- *Distinct Outputs* $\mathcal{P}^d_s$: Outputs ports that are not inputs of any subsystems.

The union of the two related inputs forms the related ports of the feedback composition, denoted as $\mathcal{P}^r = \mathcal{P}^r_{s_1} \cup \mathcal{P}^r_{s_2}$. Fig. 5.8 (b) illustrates the port partition for the example systems.

5.3.2 Problem Definition

This section first provides a motivating example to show the vacuous implementation issue in feedback composition. Then we formally define the verification problem for ensuring strong replaceability in feedback composition.

5.3.2.1 Motivating Example

We utilize a feedback amplifier as our motivating example. Feedback amplifiers are widely used components in circuit and controller design, known for their ability to improve stability and reduce variability. As depicted in Fig, 5.9 (a), a feedback amplifier consists of two subsystems: an amplifier with an open loop gain A_{OL} and a feedback network with a feedback factor β. The behavior of the feedback amplifier can be described as $y = \frac{A_{OL}}{1+A_{OL}\beta}x$, where $\frac{A_{OL}}{1+A_{OL}\beta}$ represents the closed loop gain of the feedback amplifier.

Consider a specification requiring a closed loop gain between 86 and 96. Designers may write the system contract $\mathcal{C}_s = (A_s, G_s)$ and propose subsystem contracts, as shown in Fig. 5.9 (b), for the internal amplifier ($\mathcal{C}_1 = (A_1, G_1)$) and the feedback network ($\mathcal{C}_2 =$

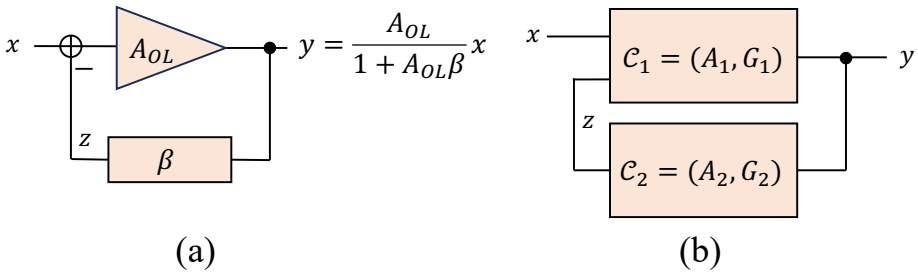


Figure 5.9: Illustration of the motivating example based on (a) a system of feedback amplifier and (b) the contracts for representing the system.

(A_2, G_2)) as $\mathcal{C}_s = (True, 86x \leq y \leq 96x)$, $\mathcal{C}_1 = (True, 95(x - z) \leq y \leq 105(x - z))$ and $\mathcal{C}_2 = (True, z = 0.001y)$, where the open loop gain A_{OL} ranges between 95 and 105, while β is fixed at 0.001.

The composition of the subsystem contracts refines the system contract, as the closed loop gain $\frac{A_{OL}}{1+A_{OL}\beta}$ is bounded by $[86.758, 95.023]$. The range of A_{OL} provides flexibility for designers' decisions and allows variation in actual implementation.

However, the following refinement $\mathcal{C}_1' = (A_1', G_1')$ for subsystem contract $\mathcal{C}_1$ results in empty desired behaviors in the composition $\mathcal{C}_1' \parallel \mathcal{C}_2 = (True, \emptyset)$:

$$A_1' = True, G_1' = \begin{cases} y = 95(x - z), & z \geq 0.09x \\ y = 105(x - z), & z < 0.09x. \end{cases}$$

We can observe that no values for x, y, and z can satisfy both G_1' and G_2, indicating that the refinement violates the strong replaceability of the system contract and leads to a vacuous implementation. Although $\mathcal{C}_1'$ refines the subsystem contracts and its composition with $\mathcal{C}_2$ satisfies the refinement relation, our design goal is not met as no behaviors are guaranteed given any inputs. As a result, the decomposition into $\mathcal{C}_1$ and $\mathcal{C}_2$ is problematic as it may lead to $\mathcal{C}_1'$ during independent development and cause vacuous implementation.

This absence of desired behaviors reflects that the composed system's behavior cannot be captured at the current abstraction level of the amplifier's physical model. In practice, such a closed-loop amplifier would produce unstable output values.

5.3.2.2 Strong Replaceability in Feedback Composition

The example demonstrates that individual contract refinement in feedback composition does not always guarantee strong replaceability to the system contract, even if all contracts are receptive. Motivated by the example, it is crucial to know under what conditions strong replaceability holds for any receptive refinement of the subsystem contracts, and how we can verify strong replaceability given a system contract and subsystem contracts.

Consequently, we define the verification problem of strong replaceability in feedback composition as follows:

Problem 5.1. *Given a receptive system contract $\mathcal{C}_s$ and receptive subsystem contracts $\mathcal{C}_1$ and $\mathcal{C}_2$ such that $\mathcal{C}_s \succeq \mathcal{C}_1 \parallel \mathcal{C}_2$, check whether the composition of any receptive refinement of the subsystem contracts $\mathcal{C}_1' \preceq \mathcal{C}_1$ and $\mathcal{C}_2' \preceq \mathcal{C}_2$ strongly replaces the system contract.*

In the following sections, we shall develop the theory about the conditions and present the proposed algorithms to answer the two questions and address the verification problem.

5.3.3 Fixed Obligations in AG Contracts

Strong replaceability in feedback composition, by definition, requires that any receptive refinement to its subsystem contracts results in a composition allowing at least one behavior for every targeted environment. Therefore, the desired behaviors in the composed subsystem contracts and their potential changes after refinement are critical for addressing Problem 5.1.

For every targeted environment, if both subsystem contract refinements preserve a behavior with the same value for their related inputs and outputs, this behavior ensures that the desired behaviors remain non-empty, thus satisfying strong replaceability. When both contracts permit exactly one behavior for the targeted environment, receptiveness guarantees that this behavior is preserved, meeting the requirement of strong replaceability for the targeted environment. However, when the contracts allow multiple resulting behaviors under the targeted environment, ensuring strong replaceability becomes more challenging to verify.

This section aims to formalize and visualize the impact of multiple resulting behaviors. First, *fixed obligations* in feedback compositions of contracts are defined as the common behaviors for the related inputs and outputs. Fixed obligations are analogous to fixed points, as the behaviors in feedback compositions of systems correspond to fixed points [98]. Building on the concept of fixed obligations, the *fixed obligation graph* is introduced to illustrate the relationships between multiple behavior choices and fixed obligations, as well as to analyze potential changes under refinement. The fixed obligation graph forms the foundation for determining strong replaceability.

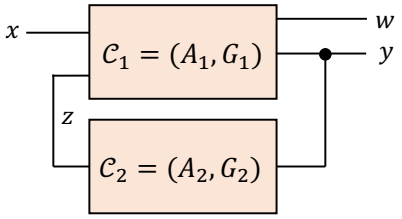
5.3.3.1 Fixed Obligations in Contract Feedback Composition

Fixed obligations in a feedback composition of contracts are defined as follows:

Definition 5.4. *Given contracts $\mathcal{C}_1 = (A_1, G_1)$ and $\mathcal{C}_2 = (A_2, G_2)$, the set of fixed obligations of the composition, denoted as $\mathcal{F}_{\mathcal{C}_1,\mathcal{C}_2}$, is defined as $\mathcal{F}_{\mathcal{C}_1,\mathcal{C}_2} = \pi_{\mathcal{P}_s}(G_1 \cap A_1) \cap \pi_{\mathcal{P}_s}(G_2 \cap A_2)$.*

A behavior $e \in \mathcal{B}_{\mathcal{P}_s}$ is a fixed obligation of $\mathcal{C}_1$ and $\mathcal{C}_2$ if $e \in \mathcal{F}_{\mathcal{C}_1,\mathcal{C}_2}$. Intuitively, fixed obligations are formed by the desired behaviors $G_1 \cap A_1$ and $G_2 \cap A_2$, subject to the constraint that the system inputs must match the targeted environment behavior. The projection to $\mathcal{P}_s$ ensures the desired behaviors are described using the system ports.

We also define the fixed obligation set under a targeted environment behavior $e \in \mathcal{B}_{\mathcal{P}_{I_s}}$ as $\mathcal{F}_{\mathcal{C}_1,\mathcal{C}_2}(e) = \pi_{\mathcal{P}^r}(\mathcal{F}_{\mathcal{C}_1,\mathcal{C}_2} \cap \pi_{\mathcal{P}_s}(\{e\}))$. Here the projection to $\mathcal{P}^r$ produces the result focusing



$x \in \mathbb{Z}$
$y \in \{-1, -2, 0, 1, 2, 3, 4\}$
$z \in \{-1, 0, 1, 2\}$
$w \in \mathbb{Z}$

$A_s: x = 2 \quad G_s: ((y = x|y-1| - x) \vee (y = |y-1| - 1 + x)) \wedge (w = 2x)$
$A_1: True \quad G_1: \big((y = xz) \vee (y = x + z)\big) \wedge (w = 2x)$
$A_2: True \quad G_2: z = |y-1| - 1$

Figure 5.10: An example contract and subsystem contracts for illustrating the fixed obligations and fixed obligation graph.

on the related ports, given that the targeted environment is clear in the context and the values of distinct outputs do not affect the fixed obligations.

The following example illustrates the concept of fixed obligations:

Example 5.4. *Let $\mathcal{C}_s$ be the system contract, $\mathcal{C}_1$ and $\mathcal{C}_2$ be the subsystem contracts, as shown in Fig. 5.10. The fixed obligation set $\mathcal{F}_{\mathcal{C}_1,\mathcal{C}_2}$ of the composition of the subsystem contracts is* $\left\{ (x,y,z,w) \in \mathcal{B}_{\mathcal{P}_s} \;\middle|\; \begin{array}{l} ((y = xz) \vee (y = x + z)) \wedge (w = 2x) \wedge \\ (z = |y-1| - 1) \end{array} \right\}$.

For a targeted environment $x = 2$, the fixed obligation set under the targeted environment is

$$\begin{aligned} \mathcal{F}_{\mathcal{C}_1,\mathcal{C}_2}(2) &= \pi_{\mathcal{P}^r}(\mathcal{F}_{\mathcal{C}_1,\mathcal{C}_2} \cap \pi_{\mathcal{P}_s}(\{e\})) \\ &= \pi_{\mathcal{P}^r}(\{(2,0,0,4)_{x,y,z,w}, (2,1,-1,4)_{x,y,z,w}, (2,2,0,4)_{x,y,z,w}, \\ &\qquad (2,3,1,4)_{x,y,z,w}, (2,4,2,4)_{x,y,z,w}\}) \\ &= \{(0,0)_{y,z}, (1,-1)_{y,z}, (2,0)_{y,z}, (3,1)_{y,z}, (4,2)_{y,z}\}, \end{aligned}$$

where the subscript of the tuples denotes the order of port that the value corresponds to: $(1,-1)_{y,z}$ means $y = 1$ and $z = -1$.

5.3.3.2 Unstable Fixed Obligation

As previously introduced, contract refinement results in a subset of the original desired behavior set within the targeted environment, while receptiveness requires a non-empty desired behavior set for every targeted environment. The resulting behaviors that do not form a fixed obligation may cause the disappearance of the fixed obligation after refinement. Consequently, in such cases, a refinement can be constructed to eliminate the behaviors forming a fixed obligation while satisfying the receptiveness by preserving the other behaviors, leading

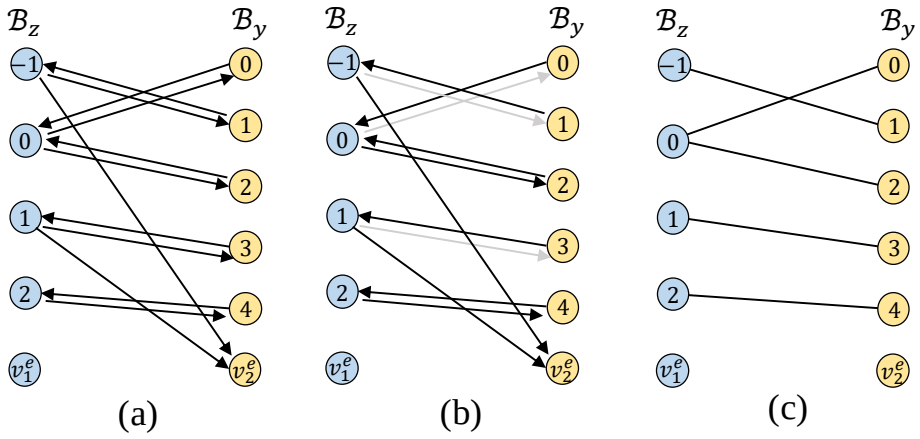


Figure 5.11: Illustration of (a) the fixed obligation graph for Example 5.4, (b) an example of its receptive subgraph by performing a receptive refinement on contracts $\mathcal{C}_1$, and (c) the four component graphs formed by its strongly connected components.

to the disappearance of the fixed obligation. Therefore, we categorize the fixed obligation with this property as *unstable fixed obligations*:

Definition 5.5. *Given system targeted environment $e \in \mathcal{B}_{\mathcal{P}_{I_s}}$ and subsystem contracts $\mathcal{C}_1 = (A_1, G_1)$ and $\mathcal{C}_2 = (A_2, G_2)$, a fixed obligation $e_f \in \mathcal{F}_{\mathcal{C}_1,\mathcal{C}_2}(e)$ is an unstable fixed obligation if $B_{\mathcal{C}_1}(e, e_f) \not\subseteq F_{\mathcal{C}_1,\mathcal{C}_2}(e)$ or $B_{\mathcal{C}_2}(e, e_f) \not\subseteq F_{\mathcal{C}_1,\mathcal{C}_2}(e)$, where*

$$B_{\mathcal{C}_1}(e, e_f) = \pi_{\mathcal{P}^r}(A_1 \cap G_1 \cap \pi_{\mathcal{P}_{s_1}}(\{e\}) \cap \pi_{\mathcal{P}^r}(\pi_{\mathcal{P}_1^r}(e_f)))$$
$$B_{\mathcal{C}_2}(e, e_f) = \pi_{\mathcal{P}^r}(A_2 \cap G_2 \cap \pi_{\mathcal{P}_{s_2}}(\{e\}) \cap \pi_{\mathcal{P}^r}(\pi_{\mathcal{P}_2^r}(e_f)))$$

are the desired behaviors of each subsystem contract when the system input values are from the targeted environment e and the related input values are from fixed obligation e_f.

The following example illustrates the concept of unstable fixed obligation using the contracts in Example 5.4:

Example 5.5. *Under the targeted environment $x = 2$, the fixed obligations $(3,1)_{y,z}$ and $(1,-1)_{y,z}$ are unstable fixed obligations, while the other fixed obligations are not. For $(3,1)_{y,z}$, $B_{\mathcal{C}_1}((2)_x, (3,1)_{y,z}) = \{(3,1)_{y,z}, (2,1)_{y,z}\}$ but $(2,1)_{y,z}$ is not a fixed obligation. Similarly, for $(1,-1)_{y,z}$, $B_{\mathcal{C}_1}((2)_x, (1,-1)_{y,z}) = \{(-2,-1)_{y,z}, (1,-1)_{y,z}\}$ but $(-2,-1)_{y,z}$ is not a fixed obligation.*

5.3.3.3 Fixed Obligation Graph

Intuitively, we can view the refinement of two subsystems A and B as a game to create refinement that violates strong replaceability. In the game, Subsystem B tries to trick Subsystem A by creating behaviors that do not match the related input values of Subsystem A.

Given a system input, if Subsystem B can only produce behaviors that match the related input values for Subsystem A, the behaviors are "ensured" and meet strong replaceability, indicating that Subsystem B loses the game. If Subsystem B can always create behaviors that do not match the related input of A, Subsystem B wins and results in an empty desired behavior that violates strong replaceability.

Therefore, reasoning about the possible fixed obligations in refinement is crucial for ensuring strong replaceability. We thus propose *fixed obligation graph*, which contains all fixed obligations and the relevant desired behaviors under a given system's targeted environment to help visualize their relations.

Given a system's targeted environment $e_s \in \mathcal{B}_{\mathcal{P}_{I_s}}$ and subsystem contracts $\mathcal{C}_1$ and $\mathcal{C}_2$, a fixed obligation graph $G^{e_s}_{\mathcal{C}_1,\mathcal{C}_2} = (V_1, V_2, E)$ is a directed bipartite graph. The vertex set V_1 and V_2 correspond to the behaviors in related inputs $\mathcal{B}_{\mathcal{P}^r_{s_1}}$ and $\mathcal{B}_{\mathcal{P}^r_{s_2}}$, respectively. Each vertex set contains regular vertices and an external behavior vertex. Every regular vertex represents a related input behavior forming a fixed obligation, while the external behavior vertex encompasses all related input behavior in a non-fixed obligation behavior. We denote the two external behavior vertices as $v^e_1 \in V_1$ and $v^e_2 \in V_2$. An edge $e = (v_a, v_b) \in E$ in the graph means that the related input behaviors v_a can produce v_b from the desired behavior of a subsystem contract. If both vertices are regular, the combined behavior from the vertices is a fixed obligation.

Fig. 5.11 (a) shows the fixed obligation graph for Example 5.4 under a system's target environment $x = 2$. From the fixed obligation graph, we can also observe that the edge representing an unstable fixed obligation must have at least one of its vertices connected to an external behavior vertex, as illustrated by $(3, 1)_{y,z}$ and $(1, -1)_{y,z}$ in the example.

Through fixed obligation graphs, we can observe the changes in desired behaviors and fixed obligations. Any refinement to a subsystem contract removes some edges starting from its related input behaviors. Receptive refinement ensures that every regular vertex of its related inputs has at least one outward edge. Individual refinement on both subsystem contracts thus results in a subgraph $G^e_{\mathcal{C}_1,\mathcal{C}_2} = (V_1, V_2, E')$ with the same sets of vertices and a subset of edges from the original graph. The remaining fixed obligations after refinement are the behavior combined by related inputs v_a and v_b such that $(v_a, v_b) \in E'$ and $(v_b, v_a) \in E'$. The subgraph is called a receptive subgraph if all regular vertices have at least one outward edge. Fig. 5.11 (b) provides an example of a receptive subgraph derived from Fig. 5.11 (a).

5.3.4 Conditions for Strong Replaceability

As a fixed obligation graph contains fixed obligations and the relevant desired behaviors, their relationship allows us to convert Problem 5.1 to an equivalent problem using the concept of a fixed obligation graph:

Problem 5.2. *Given a receptive system contract $\mathcal{C}_s = (A_s, G_s)$ and subsystem receptive contracts $\mathcal{C}_1 = (A_1, G_1)$ and $\mathcal{C}_2 = (A_2, G_2)$, verify if for all system's targeted environment*

$e \in A_s$, all receptive subgraphs of the fixed obligation graph $G^e_{\mathcal{C}_1,\mathcal{C}_2}$ have at least one fixed obligation.

Although Problem 5.2 asks whether all receptive subgraphs have fixed obligations, we can decompose the problem by considering strongly connected components in a fixed obligation graph. Strongly connected components can independently ensure a fixed obligation, as the fixed obligations and desired behaviors resulting from edge selections in a strongly connected component do not affect those in other strongly connected components. We refer to the fixed obligations in a strongly connected component as *fixed obligation groups*. For example, in Fig. 5.11 (a), the fixed obligation graph contains four fixed obligation groups: $\{(0,0)_{y,z}, (2,0)_{y,z}\}$, $\{(1,-1)_{y,z}\}$, $\{(1,3)_{y,z}\}$, and $\{(4,2)_{y,z}\}$. We can convert the strongly connected component into an undirected graph by replacing the two directed edges of every fixed obligation with an undirected edge. We call the undirected graph *component graph*. Figure 5.11 (c) shows all component graphs formed by the strongly connected components in Fig. 5.11 (a).

With the observation, we can address Problem 5.2 by identifying the condition for a fixed obligation group to ensure fixed obligations. This section presents such conditions by observing properties in fixed obligation graphs and then summarizes the conditions for solving Problems 5.1 and 5.2.

5.3.4.1 Conditions for Fixed Obligation Group

To determine whether a fixed obligation group ensures fixed obligations after receptive refinement, we develop several theorems and combine them as the conditions for a fixed obligation group to ensure fixed obligations.

First, the following theorem states that a fixed obligation group containing unstable fixed obligations cannot ensure a fixed obligation.

Theorem 5.6. *If a fixed obligation group contains an unstable fixed obligation, there exists a receptive subgraph in which no fixed obligation remains.*

Proof. To construct such a receptive subgraph, we begin by creating a spanning tree on the component graph containing the fixed obligation group. We then introduce one edge between the external behavior vertex and one of the vertices whose behavior forms the unstable fixed obligation. Finally, we assign direction to the edges based on the shortest path to the external behavior vertex through the spanning tree. The resulting graph is a subgraph according to the definition of the spanning tree and is receptive as every regular vertex connects to one vertex. □

Fig. 5.12 (a) shows a fixed obligation graph satisfying the condition of this theorem and one of its receptive subgraphs, which has no fixed obligations.

The following theorem describes the condition when fixed obligations are ensured in a fixed obligation group without unstable fixed obligations:

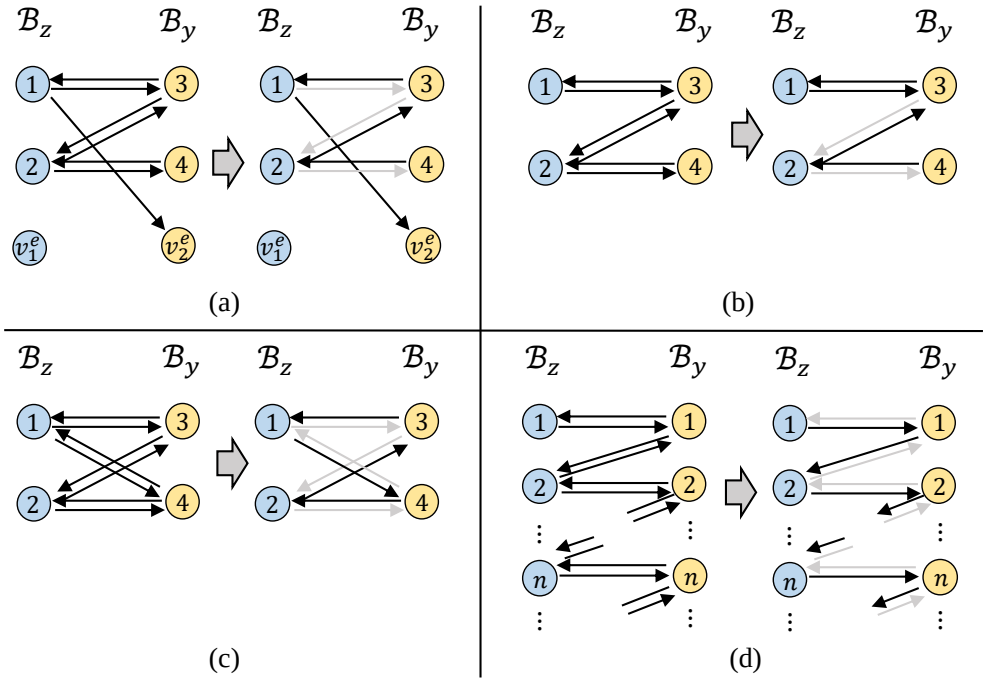


Figure 5.12: Examples of fixed obligation graphs illustrate the results in cases specified by the theorems. Each subfigure presents an original fixed obligation graph on the left and its receptive subgraph on the right, showing no fixed obligation for (a), (c), and (d), while (b) provides an example demonstrating that it ensures at least one fixed obligation.

Theorem 5.7. *A fixed obligation group containing $n-1$ fixed obligations and no unstable fixed obligations from a strongly connected component with n vertices ensures at least one fixed obligation remains in any receptive refinement.*

Proof. For a strongly connected component with n vertices, its receptive subgraphs must have at least n edges. However, since the fixed obligation group contains $n-1$ different fixed obligations, there are only $n-1$ edges with distinct endpoints. According to the pigeonhole principle, at least two edges share the same endpoints with opposite directions, resulting in a behavior that forms a fixed obligation. □

The theorem implies that a component graph ensures a fixed obligation if it is a tree with n vertices and does not have infinite paths, as such a component graph must contain $n-1$ edges, each representing a fixed obligation. The remaining cases are component graphs with cycles or infinite paths. Figure 5.12 (b) shows a fixed obligation graph satisfying the condition of the theorem. With four vertices and three pairs of edges in the graph, receptiveness ensures that at least two edges with opposite directions and shared endpoints are preserved in the subgraph.

The following theorem shows that those cases do not guarantee fixed obligations:

Theorem 5.8. *If a component graph contains a cycle or an infinite path, there exists a receptive subgraph in which no fixed obligation remains.*

Proof. To construct such a receptive subgraph, we first remove all the edges forming the loop or the infinite path. Next, we connect each vertex involved in the cycle or the infinite path to the external behavior vertex, forming a new undirected graph $\tilde{G}$. We then follow the steps in the proof for Theorem 5.6 to construct a receptive graph based on $\tilde{G}$. Finally, we remove the edges connecting to the external behavior vertex and reintroduce the directed edges for one direction of the loop or the infinite path.

The resulting graph, by construction, does not have any fixed obligations since only one directed edge is chosen for every fixed obligation. Additionally, it is a subgraph since the added edges are eventually removed. Its receptiveness is guaranteed by the steps from the proof of Theorem 5.6 and the last step, as every vertex has an outward edge. □

Figure 5.12 (c) and (d) depict fixed obligation graphs containing a cycle and an infinite path, respectively. For the graph with a cycle, the cycle can be leveraged to construct a receptive graph having no fixed obligations. Similarly, the structure of an infinite path can be utilized to create a receptive graph without fixed obligations.

Combining the above theorems, we can derive the conditions for a fixed obligation group to ensure fixed obligations in receptive refinement:

Theorem 5.9. *A fixed obligation group guarantees at least one fixed obligation if and only if all of the following conditions hold: (1) The fixed obligation group does not contain any unstable fixed obligation, and (2) The corresponding component graph does not contain cycles or infinite paths.*

For instance, in Figure 5.11 (a), the fixed obligation group $\{(0,0)_{y,z}, (2,0)_{y,z}\}$ and $\{(4,2)_{y,z}\}$ satisfies the conditions, and thus the subsystem contracts ensure fixed obligations under the targeted environment $x = 2$.

5.3.4.2 The Necessary and Sufficient Conditions

We now present the necessary and sufficient conditions for ensuring strong replaceability in feedback compositions.

Theorem 5.10. *Let $\mathcal{C}_s = (A_s, G_s)$ be a receptive system contract, and $\mathcal{C}_1 = (A_1, G_1)$ and $\mathcal{C}_2 = (A_2, G_2)$ be its subsystem receptive contracts. The feedback composition ensures strong replaceability if and only if, for every targeted environment $e_s \in A_s$, the fixed obligation graph $G^{e_s}_{\mathcal{C}_1,\mathcal{C}_2}$ contains at least a fixed obligation group that satisfies the conditions in Theorem 5.9.*

Once every targeted environment has a fixed obligation, the behavior for each targeted environment is ensured, which guarantees the satisfaction of strong replaceability to the system contract.

We can understand the conditions using the intuition game between Subsystems A and B. At the beginning of the game, Subsystem A produces a behavior satisfying its desired behaviors and the system input. Given the behavior, Subsystem B tries to produce a different behavior that does not match A's behaviors. Subsystem A then produces another behavior based on the new inputs provided by Subsystem B. This process continues, and the sequence of the behaviors can be visualized as a path on the graph. If this graph forms a tree structure with fixed obligations, subsystem B will eventually fail because it will be forced to produce the same behaviors at the ends of the tree branches. However, if there are unstable fixed obligations, cycles, or infinite paths, subsystem B can exploit them to never return to the matched behavior.

We now revisit our motivating example in Section 5.3.2.1 to examine the conditions. Considering the targeted environment $x = 1$, the behavior $(95.023, 0.095023)_{y,z}$ is a fixed obligation satisfying both G_1 and G_2. However, the desired behaviors of the related inputs z for G_1 contain $(85.9728, 0.095023)_{y,z}$, which is not a fixed obligation. As a result, $(51.29, 0.5129)_{y,z}$ represents an unstable fixed obligation. Furthermore, all fixed obligations under the environment $x = 1$ belong in a fixed obligation group as we can produce them by applying G_1 and G_2 with the available options for A_{OL}. Therefore, the condition for strong replaceability is not met, and the refinement G_1' is an example that violates strong replaceability.

5.3.5 Proposed Algorithms

With the theoretical foundation from the conditions for ensuring strong replaceability, this section proposes two algorithms to verify the condition and address the verification problem in Problem 5.1. The first algorithm explores the fixed obligations and their desired behaviors to identify the fixed obligation group that satisfies the conditions. However, it is limited to finite behavior sets, requiring bounded port types. To overcome this limitation, we propose a second algorithm that encodes the conditions into a series of quantified satisfiability modulo theory (SMT) problems [12]. This approach allows us to leverage background theories for verification. To maintain the generality of the algorithms, we refrain from relying on specific background theories within the algorithms, assuming that the solvers can effectively handle our problem. While incorporating specific background theories may potentially improve the algorithm, such endeavors are beyond the scope of this paper. Future research will be essential to adapt the algorithms for certain applications and background theories.

5.3.5.1 Algorithms for Finite Sets

Algorithm 1 outlines the procedure to verify the strong replaceability of a feedback composition with contracts defined over finite behavior sets. The algorithm iteratively picks a fixed obligation, collects all fixed obligations belonging to the same fixed obligation group as the fixed obligation, and verifies if the group satisfies the condition in Theorem 5.9 for every targeted environment. Initially, the candidates for exploration are all fixed obligations. Lines

4–5 select a fixed obligation that has not yet been encountered in any explored fixed obligation group until all fixed obligations have been explored. Lines 6 collect the fixed obligation group where the fixed obligation belongs and verify the conditions. Lines 7–8 terminate the inner loop and proceed to the next targeted environment if the fixed obligation group ensures a fixed obligation. Otherwise, Line 9 removes the fixed obligations of the group from candidates to avoid repetition. Once we find a targeted environment without any fixed obligation group satisfying the conditions, Line 11 concludes that the strong replaceability does not hold. Conversely, if all targeted environments have a fixed obligation group satisfying the conditions, Line 12 concludes that strong replaceability holds.

The steps for collecting the fixed obligation group and verifying the conditions are outlined in Algorithm 2. Since each fixed obligation corresponds to an edge in the component graph, we traverse the component graph by the desired behaviors from the related inputs of a fixed obligation, using the intersection of the related inputs with the subsystem contract guarantees. Moreover, the bipartite nature of the fixed obligation graph enables the search to explore neighboring vertices and edges by alternatively applying the desired behaviors from $\mathcal{C}_1$ and $\mathcal{C}_2$. Lines 4–11 perform the search by applying desired behaviors from $\mathcal{C}_1$, while Lines 12–19 do so from $\mathcal{C}_2$. Lines 7–8 and 15–16 indicate that the conditions are not met if an unstable fixed obligation or a cycle is detected, An unstable fixed obligation is identified if the desired behaviors are not a subset of the fixed obligations, while a cycle is detected if an explored fixed obligation is encountered during the search. The process continues until no new fixed obligation can be explored from the component graph. Finally, the fixed obligation group and the satisfaction of the conditions are returned.

Since the behavior sets are finite and each fixed obligation is explored once, the algorithm is guaranteed to terminate within at most $|\mathcal{F}|$ iterations of fixed obligation group collection. Furthermore, any unstable fixed obligations or cycles are detected through the breadth-first search, and an infinite path must not exist due to the finite sets. Therefore, the algorithm is complete and sound for finite sets. Its complexity is $O(|\mathcal{B}_{\mathcal{P}_I}||\mathcal{B}_{\mathcal{P}^r}|^2)$, as each fixed obligation is explored once, and for every fixed obligation, its desired behaviors are explored for checking the conditions.

5.3.5.2 Algorithms for Infinite Sets

In many applications, such as circuit designs [126] or control systems [22], components often involve port types in infinite sets, which cannot be handled by the above algorithm. These infinite sets are usually compactly described by equations or predicates. To overcome the limitation, we propose the second algorithm based on a series of quantified SMT problems, assuming the supported theory solver can effectively reason about the descriptions. As summarized in Fig. 5.13, the algorithm iteratively performs two main steps: *Positive Proof* and *Negative Proof*. Both steps utilize an upper bound depth d for constraining the traversal depth for component graphs. Increasing the upper bound depth affects the complexity of the SMT formula as more clauses and variables are involved. Under the traversal depth constraint, positive proof attempts to prove that every targeted environment has a fixed obli-

Algorithm 1 Strong Replaceability for Finite Set Contracts

Inputs: A system contract $\mathcal{C}_s = (A_s, G_s)$, subsystem contracts $\mathcal{C}_1 = (A_1, G_1)$, $\mathcal{C}_2 = (A_2, G_2)$, and their port partition $\mathcal{P}_{I_s}$, $\mathcal{P}^r_{s_1}$, $\mathcal{P}^r_{s_2}$, and $\mathcal{P}^d_s$

Output: Whether strong replaceability holds for the feedback composition

for all $e \in \pi_{\mathcal{P}_{I_s}}(A_s)$ **do**
  $\mathcal{F} \leftarrow \mathcal{F}_{\mathcal{C}_1,\mathcal{C}_2}(e)$ {fixed obligation set under e}
  candidates $\leftarrow \mathcal{F}$, *sat* $\leftarrow$ False
  while *candidates* $\neq \emptyset$ **do**
    root $\leftarrow$ `get_one_element`(*candidates*)
    group, *sat* $\leftarrow$ `collect_group_and_verify`(*root*, e)
    if *sat* **then**
      break
    candidates $\leftarrow$ *candidates* $-$ *group*
  if *sat* == False **then**
    return False
return True

Algorithm 2 `collect_group_and_verify`

Inputs: Subsystem contracts $\mathcal{C}_1 = (A_1, G_1)$, $\mathcal{C}_2 = (A_2, G_2)$, their port partition $\mathcal{P}_{I_s}$, $\mathcal{P}^r_{s_1}$, $\mathcal{P}^r_{s_2}$, and $\mathcal{P}^d_s$, a fixed obligation *root*, , a targeted environment e, and the fixed obligation set $\mathcal{F}$ under e.

Output: The fixed obligation group containing *root* and whether the group satisfies the condition to ensure fixed obligations

group $\leftarrow \{root\}$, *sat* $\leftarrow$ True, *explore_c1* $\leftarrow$ {root}, *explore_c2* $\leftarrow \{root\}$
while *explore_c1* $\neq \emptyset \vee$ *explore_c2* $\neq \emptyset$ **do**
  next_explore_c1 $\leftarrow \emptyset$, *next_explore_c2* $\leftarrow \emptyset$
  for all $e_f \in$ *explore_c1* **do**
    neighbors $\leftarrow$ desired behaviors of $\mathcal{C}_1$ under inputs $\pi_{\mathcal{P}^r_{s_1}}(e_f)$ and e
    for all $e_n \in$ (*neighbors*$/\{e_f\}$) **do**
      if $e_n \in$ *group* $\vee$ $e_n \notin \mathcal{F}$ **then**
        sat $\leftarrow$ False
      else
        group $\leftarrow$ *group* $\cup \{e_n\}$
        next_explore_c2 $\leftarrow$ *next_explore_c2* $\cup \{e_n\}$
  for all $e_f \in$ *explore_c2* **do**
    neighbors $\leftarrow$ desired behaviors of $\mathcal{C}_2$ under inputs $\pi_{\mathcal{P}^r_{s_2}}(e_f)$ and e
    for all $e_n \in$ (*neighbors*$/\{e_f\}$) **do**
      if $e_n \in$ *group* $\vee$ $e_n \notin \mathcal{F}$ **then**
        sat $\leftarrow$ False
      else
        group $\leftarrow$ *group* $\cup \{e_n\}$
        next_explore_c1 $\leftarrow$ *next_explore_c1* $\cup \{e_n\}$
  explore_c1 $\leftarrow$ *next_explore_c1*
  explore_c2 $\leftarrow$ *next_explore_c2*
return *group*, *sat*

gation group satisfying conditions in Theorem 5.9, while negative proof is designed to prove that no fixed obligation group under some targeted environments satisfies the conditions.

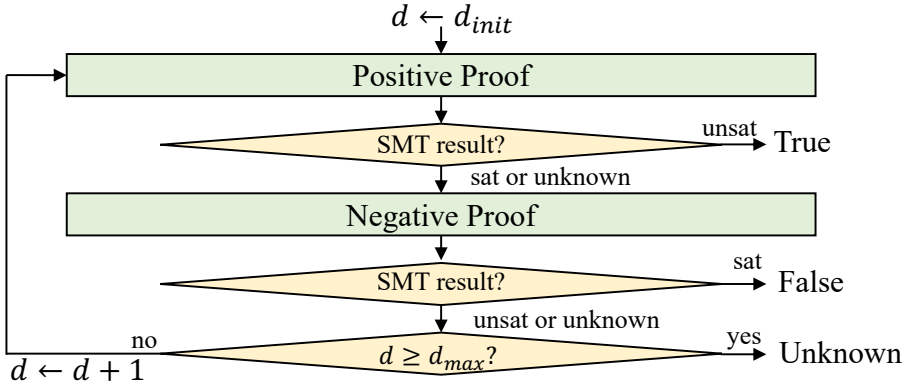


Figure 5.13: An overview of the proposed algorithm for verifying strong replaceability for infinite set contracts.

Algorithm 3 Positive Proof

Inputs: A system contract $\mathcal{C}_s = (A_s, G_s)$, subsystem contracts $\mathcal{C}_1 = (A_1, G_1)$, $\mathcal{C}_2 = (A_2, G_2)$, their port partition $\mathcal{P}_{I_s}$, $\mathcal{P}^r_{s_1}$, $\mathcal{P}^r_{s_2}$, and $\mathcal{P}^d_s$, and the traversal depth d

Output: Whether the system contract has an environment with no fixed obligation group satisfying the condition within traversal depth d

Copy the variables for related ports $\mathcal{P}^r$[i] and distinct outputs $\mathcal{P}^d_s[i]$
Encode clauses $c^i_{\mathcal{F}}$, c_{A_s}, $c^i_{A_1}$, $c^i_{A_2}$, $c^i_{G_1}$, $c^i_{G_2}$
`// i: encoded by the (i)th related port values`
Encode clauses c^i_{eq1} and c^i_{eq2} for equivalent related input value
`// i: the ith and (i-1)th` $\mathcal{P}^r_{s_1}$ `value are the same for` c^i_{eq1}`, and the` $\mathcal{P}^r_{s_2}$ `values for` c^i_{eq2}
for $k = 1 \dots d$ **do**
    $c^k_{n1} = c^k_{eq1} \wedge (\exists \mathcal{P}^d_s[k], c^k_{G_1})$
    $c^k_{n2} = c^k_{eq2} \wedge (\exists \mathcal{P}^d_s[k], c^k_{G_2})$
$c^0_{tree1} = \forall \mathcal{P}^r[d], c^d_{n1} \implies c^d_{eq2}$
$c^0_{tree2} = \forall \mathcal{P}^r[d], c^d_{n2} \implies c^d_{eq1}$
for $k = 1 \dots d-1$ **do**
    $c^k_{tree1} = (\forall \mathcal{P}^r[d-k], c^{d-k}_{n1} \implies (c^{d-k}_{\mathcal{F}} \wedge (c^{k-1}_{tree2} \vee c^{d-k}_{eq2})))$
    $c^k_{tree2} = (\forall \mathcal{P}^r[d-k], c^{d-k}_{n1} \implies (c^{d-k}_{\mathcal{F}} \wedge (c^{k-1}_{tree1} \vee c^{d-k}_{eq1})))$
$c_{pos} = c_{A_s} \wedge (\forall (\mathcal{P}^r[0], \mathcal{P}^d_s[0]), \neg\, c^0_{\mathcal{F}} \vee \neg\, c^{d-1}_{tree1} \vee \neg\, c^{d-1}_{tree2})$
return `solve_SMT(`c_{pos}`)`

The algorithm begins with an initial depth d_{init}. The result is immediately returned once either of the proofs can conclude the satisfaction of the conditions. Otherwise, the traversal depth d is incremented for the next iteration of proofs until a conclusive result is reached or the maximum depth d_{max} is exceeded. In the latter case, the algorithms return unknown for strong replaceability. To prevent the risk of vacuous implementations, such subsystem contracts should not be used for independent development.

The positive proof, outlined in Algorithm 3, encodes an SMT formula to find a targeted environment where no component graphs are trees within the traversal depth d. The SMT

Algorithm 4 Negative Proof

Inputs: A system contract $\mathcal{C}_s = (A_s, G_s)$, subsystem contracts $\mathcal{C}_1 = (A_1, G_1)$, $\mathcal{C}_2 = (A_2, G_2)$, their port partition $\mathcal{P}_{I_s}$, $\mathcal{P}^r_{s_1}$, $\mathcal{P}^r_{s_2}$, and the traversal depth d

Output: Whether the system contract has an environment that only contains fixed obligation groups violating the condition within traversal depth d

1: Generate all clauses as described in Algorithm 3
2: Encode c^i_{loop} for repetition of non-neighboring related port values
3: $c^0_{fail1} = (\exists \mathcal{P}^r[d],\, c^d_{n1} \wedge \neg\, c^d_{\mathcal{F}})$
4: $c^0_{fail2} = (\exists \mathcal{P}^r[d],\, c^d_{n2} \wedge \neg\, c^d_{\mathcal{F}})$
5: **for** $k = 1 \dots d-1$ **do**
6: $\quad c^k_{fail1} = (\exists \mathcal{P}^r[d-k],\, c^{d-k}_{n1} \wedge \neg c^{d-k}_{eq2} \wedge (\neg\, c^{d-k}_{\mathcal{F}} \vee c^{k-1}_{fail2} \vee c^{d-k}_{loop}))$
7: $\quad c^k_{fail2} = (\exists \mathcal{P}^r[d-k],\, c^{d-k}_{n2} \wedge \neg c^{d-k}_{eq1} \wedge (\neg\, c^{d-k}_{\mathcal{F}} \vee c^{k-1}_{fail1} \vee c^{d-k}_{loop}))$
8: $c_{neg} = c_{A_s} \wedge (\forall(\mathcal{P}^r[0], \mathcal{P}^d_s[0]),\, c^0_{\mathcal{F}} \implies (c^{d-1}_{fail1} \vee c^{d-1}_{fail2}))$
9: **return** `solve_SMT(`c_{neg}`)`

formula involves variables for the system input ports and an array of $d+1$ related ports, representing an edge traversal on the fixed obligation graph. Lines 1–5 create these variables and generate the constraint clauses. $c^i_{\mathcal{F}}$ represents that the ith related input value is a fixed obligation, while the other constraints encode the elements in contracts that the port value needs to satisfy. Lines 6–10 then define the constraints on successive port values to ensure a legal traversal. Constraints c^i_{eq1}, c^i_{eq2} require the successive related input port values of $\mathcal{P}^r_{s_1}$ and $\mathcal{P}^r_{s_2}$, respectively, have the same value, while c^i_{n1} and c^i_{n2} specify that the successive related ports values must be the desired behaviors of $\mathcal{C}_1$ and $\mathcal{C}_2$, respectively. Lines 11–14 recursively define the constraints c^i_{tree1} and c^i_{tree2} to require the branch terminate in leaves within i traversal steps. The termination to leaves is represented using constraints c^i_{eq1} and c^i_{eq2}, while the recursive terms c^{k-1}_{tree1} and c^{k-1}_{tree2} allow the branch to traverse $k-1$ additional steps. Finally, Line 15 seeks a targeted environment where no fixed obligation can be found as a root to traverse and terminate to leaves within d steps. The unsatisfaction of the quantified SMT problem indicates that all targeted environments can find such a root, verifying the condition. Conversely, the satisfaction of the problem indicates that at least one targeted environment cannot find such a component graph, thus being inconclusive and requiring either the negative proof or an increase of the traversal depth.

The negative proof, as outlined in Algorithm 4, encodes an SMT formula to identify a targeted environment where all component graphs can be detected violating the condition within a traversal depth lower than d. Similar to the positive proof, the encoding process starts with defining constraints of legal traversal. Lines 3–7 then recursively define constraints c^i_{fail1} and c^i_{fail2} to specify the detection of violation within i traversal steps. Violations within k steps may involve finding an unstable fixed obligation through a desired behavior not belonging to fixed obligations, detecting a loop, or witnessing such violations in the next $k-1$ traversal steps. Finally, Line 9 requires a targeted environment where all component graphs violate the conditions through the traversal, suggesting that the strong replaceability fails. Therefore, the satisfaction of the quantified SMT problem indicates a vi-

Table 5.1: Experimental results for verifying the effectiveness of the algorithm.

Test Case	Strong Replaceability	Output	Depth d	Time (s)
finite_set_safe	O	O	NA	0.67
finite_set_loop	X	X	NA	0.61
finite_set_unstable	X	X	NA	0.67
finite_set_exp_1	O	O	NA	0.58
inf_set_safe	O	O	2 (Positive)	0.70
inf_set_loop	X	X	4 (Negative)	1.10
inf_set_unstable	X	X	4 (Negative)	1.09
inf_set_motivating	X	X	1 (Negative)	0.78

olation of the condition in Theorem 5.10, while unsatisfaction of the problem is inconclusive regarding strong replaceability, requiring further steps with the algorithms.

The algorithm is sound since it encodes the necessary and sufficient conditions. However, it is incomplete as the result remains unknown when all component graphs under a targeted environment contain an infinite path without other violations or when all satisfying fixed obligation groups have a diameter greater than d_{max}. In such cases, we may advise against using such subsystem contracts for independent development to avoid the risk of vacuous implementations. Further research is essential to directly detect the infinite paths from the contract descriptions.

5.3.5.3 Experiments

To demonstrate the effectiveness of our algorithms in verifying strong replaceability in feedback composition, we implemented the algorithms in Python with Z3 [48] as the SMT solver. Given the absence of previous work on strong replaceability in feedback composition, there are no existing benchmarks with verified results for evaluation. Therefore, we created several test cases that can be verified manually, in addition to the motivating example and Example 5.4. The finite set test cases were used for evaluating our finite set algorithm, while the infinite set ones were utilized for our infinite set algorithm with $d_{init} = 1$ and $d_{max} = 5$.

As shown in Table 5.1, our algorithms accurately verify the strong replaceability for all test cases. Furthermore, all results for infinite set test cases are obtained within 2 seconds, indicating that the infinite set algorithm can effectively verify strong replaceability for small test cases whose background theories support the SMT problem.

5.3.6 Conclusion

We presented the necessary and sufficient conditions to ensure strong replaceability of assume-guarantee contracts in feedback composition. The proposed fixed points and fixed point graphs assist in developing the conditions and proving their correctness. Based on the conditions, we developed the algorithms to verify the strong replaceability of contracts in

feedback composition. We argue that the robustness of the contract-based design framework must be ensured by the strong replaceability to prevent vacuous implementations. Our future works include developing software tools implementing the algorithms for various background languages such as linear temporal logic, finding conditions to detect infinite paths for the background languages, and investigating the strong replaceability in different contract formalisms.

5.4 Conclusion

This chapter presented the verification of contracts to ensure correct decomposition, facilitating independent design in contract-based design methodologies. The identified vacuous implementation problem indicates that solely relying on verifying refinement relations is insufficient to provide the required assurance. By formulating the requirement as contract replaceability, contract receptivity was shown to satisfy the requirement in single contract refinement and cascade composition. Furthermore, conditions for establishing contract replaceability in feedback composition were identified by examining the behavioral relations in the fixed-obligation graph, With the proposed verification algorithms, necessary verification techniques are provided to ensure a robust contract-based design process within the independent design paradigm.

Chapter 6

Simulation: Ensuring Alignment of Contracts with Design Intent

As discussed in Chapter 3, simulation can serve as an essential automation task for checking if the design intent has been correctly captured in the contracts. As the entry point of contract-based design, ensuring alignment between contracts and design intent is crucial for achieving the overall design goals. However, since contracts are manually written, mistakes may inevitably occur. To address this problem, this chapter proposes a contract simulation methodology to help ensure that manually written contracts align with the design intent. The simulation is conducted through *Constraint-based Simulation*, an SMT-based flow that generates behaviors guided by the constraints. The methodology further utilizes the proposed concepts of *critical behavior collections* and *critical component collections* to guide contract simulation. The resulting behaviors can reveal potentially incorrect operators, providing a systematic approach for detecting specification mistakes and enabling efficient correction. Experimental results demonstrate that the methodology efficiently simulates contracts and generates behaviors for designers within a reasonable runtime.

6.1 Introduction

Contract-based design [17, 20, 109, 151] has been proposed as a promising design methodology to address the cyber-physical system design challenges incurred by their increasing scale and heterogenuity. The methodology utilizes contracts, formal specifications with well-defined operations and relations, to allow rigorous reasoning and hierarchical abstraction of various design concerns and subsystems. By structuring the design process in this way, contract-based design reduces the design complexity and ensures a correct-by-construction design approach. A contract is defined as $\mathcal{C} = (\mathcal{E}, \mathcal{I})$, where $\mathcal{E}$ specifies the environments in which the design is expected to function correctly, and $\mathcal{I}$ defines the acceptable implementations, typically represented by the set of desired behaviors under the required environments. Operations and relations on contracts, including composition, conjunction, merging [17, 20,

132], as well as refinement and the strong replaceability introduced in Chapters 5, enable verification to ensure specification satisfaction throughout the hierarchical design process. The hierarchical design process involves progressively refining specifications by adding design details, decomposing them into subsystem specifications, and separating design concerns across multiple viewpoints. This process breaks down system design problems into manageable subsystem design problems, facilitating efficient system development by integrating the design results of individual subsystems. The correctness of the overall system is guaranteed through the satisfaction of specifications at each stage of the hierarchical design process.

Numerous automation tools and algorithms have been developed to support contract-based design, including verification techniques for contract refinement and decomposition [35, 76, 96], synthesis of contract-specified systems using component libraries [73, 74, 130, 152, 153], and reasoning tools for verifying the separation of design concerns [79]. These advancements in automation tools ensure that the resulting system implementation meets the provided design specifications. Thus, the design goals can be achieved as long as the given design specifications, such as the top-level specification and the specifications modeling actual components, accurately capture the design intent and component properties. Since these specifications serve as the entry point of contract-based design, ensuring their correctness is crucial to guaranteeing that the design goals are met.

However, since these contracts are manually written, mistakes may inevitably occur, leading to mismatches between the design intent and the specifications. These mistakes can arise due to unfamiliarity with contract semantics or typographical mistakes during manual writing or typing. For example, consider the specification of a resistor based on its static behavior: "*A resistor should obey Ohm's law with a resistance of* $R = 2$ *as long as the power it generates does not exceed* 20." If a designer is unfamiliar with assume-guarantee contracts or constraint-behavior contracts, they may misinterpret how to encode this specification correctly, resulting in one of the intuitive but incorrect encodings discussed in the introduction of Chapter 4. Additionally, typos are inevitable as no designer is immune to making occasional errors. These mistakes are difficult to detect during the design process, as each stage relies on the specifications rather than directly referencing the original design intent. If a specification fails to accurately capture the design intent, the entire process will faithfully follow the incorrect specification, resulting in an implementation that does not meet the actual design goals. To detect these mistakes, the design intent must be actively incorporated into the design process. The designer, who possesses the design intent, should review the specifications early to ensure that the semantics of the contracts correctly reflect the intended behavior. The earlier this review occurs, the less time is wasted on repeating the design process after revising the mistakes. Thus, a methodology that enables designers to examine written contracts and assists in identifying these mistakes is crucial to accommodating design errors within the design process.

Since the semantics of assume-guarantee contracts and constraint-behavior contracts are based on sets of acceptable and prohibited behaviors for the environment and the implementation, providing a set of sample behaviors that satisfy or violate a contract can help designers identify mistakes in their specifications. Generating such behaviors from contracts falls into

the category of simulation. Simulation [57] is a powerful technique for validating whether a given specification correctly captures the design intent. By observing generated behaviors, designers can verify expected output values or relationships between variables. For example, in ASIC design, RTL description serves as a specification for gate-level implementation. RTL simulation generates behaviors allowed by the RTL description, enabling designers to verify whether the RTL description correctly implements the intended algorithms and functionality. Therefore, we suggest that *contracts simulation*, which produces behaviors from the semantics of contracts, could offer a promising methodology for detecting specification mistakes by assisting designers in identifying issues through reviewing generated behaviors.

Therefore, this work presents a contract simulation methodology to help ensure that manually written contracts align with their intended design. The methodology focuses on leveraging simulation to derive behaviors from assume-guarantee and constraint-behavior contracts. The contributions of this work are summarized as follows:

- We introduce a contract simulation methodology within the contract-based design framework to assist designers in verifying whether manually written contracts align with their design intent. To the best of our knowledge, this is *the first work* to propose a methodology that addresses misalignment between contracts and design intent, as well as the first to introduce the notion of contract simulation.
- We also define the concept of *critical behavior collections*, which, when found to mismatch the design intent, can reveal potentially incorrect operators, thereby helping the designer quickly identify and fix the errors. These behaviors are also more likely to find misalignments, providing a systematic approach for detecting specification mistakes and enabling efficient correction through the identification and adjustment of the corresponding operators.
- We propose *automated component generation*, an algorithm that facilitates the generation of critical behavior collections by producing collections of contract environments and implementations. The behaviors generated from these collections are guaranteed to form critical behavior collections and thus can guide the simulation to produce critical behavior collections.
- We propose *constraint-based simulation*, an SMT-based flow that generates behaviors as simulation results guided by the contract, designer input, and the outcomes of the automated component generation process. The generated behaviors assist designers in verifying the alignment of the design intent with the contracts.
- Experimental results on the correctness and scalability of the proposed algorithms demonstrate that the methodology can accurately simulate contracts and provide behaviors for designers within a reasonable runtime.

The remainder of this chapter is organized as follows: Section 6.2 formulates the contract simulation problem and provides an overview of the methodology. Section 6.3 introduces the

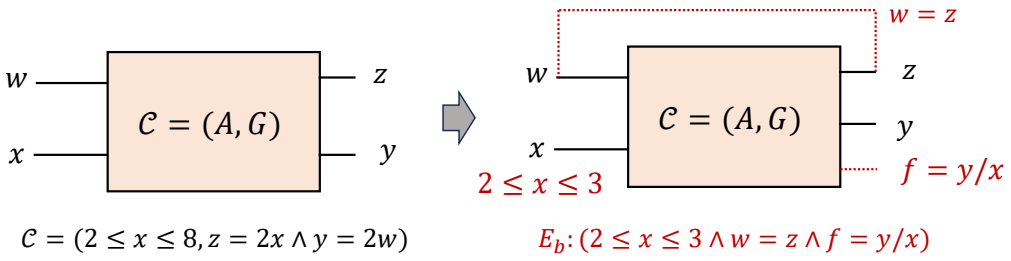


Figure 6.1: Illustration of the role of environment constraints.

automated component generation algorithm. Section 6.4 details the flow of constraint-based simulation. Section 6.5 analyzes and presents the experimental results. Finally, Section 6.6 concludes the chapter.

6.2 Contract Simulation

This section formulates the contract simulation problem and provides an overview of the proposed methodology.

6.2.1 Problem Formulation

The contract simulation problem is formulated as follows:

Problem 6.1. *Given a contract and environment constraints, find the possible resulting behaviors such that they satisfy the environment constraints and meet the contract semantics.*

Here we detail the elements of the simulation problem:

- **Contract**: A contract is represented as $\mathcal{C} = (\mathcal{E}, \mathcal{I})$ for generality. For assume-guarantee contracts, we express $\mathcal{E} = 2^A$ and $\mathcal{I} = 2^{G \cup \overline{A}}$. For constraint-behavior contract, we use $\mathcal{E} = 2^{C \cup \overline{B}}$ and $\mathcal{I} = 2^B$.

- **Environment Constraints**: Environment constraints $E_c \in \mathcal{B}$ are constraints set by the designer to guide the simulation. The constraints serve various purposes, including defining environmental inputs, connections, and relations to assist in evaluation. First, constraints can be used to enforce specific environmental inputs, such as setting the voltage for a battery or the truth value for a logic gate. Moreover, constraints can also describe connections for ports, which is useful when a designer wants to test the result under specific connection conditions. Finally, constraints can be used to define evaluation metrics based on the generated behaviors. Enabling evaluation reduces the

designer's burden by automatically extracting the resulting behaviors and performing the additional steps required to compute the metrics.

Figure 6.1 illustrates the environment constraints. The contract represents a system with four ports: w, x, y, z. The constraint $2 \leq x \leq 3$ represents the environment input, restricting the resulting behaviors to values of x between 2 and 3. The constraint $w = z$ represents a connection between ports w and z , as it forces the two ports to have the same value, simulating the behavior of connected ports. Finally, the constraint $f = y/x$ defines a relation for evaluation, with f being included in the behavior of the simulation result as the ratio of y to x. The environment constraints E_b are formed by combining these constraints through set intersections, which are equivalent to conjunctions for sets expressed in first-order logic.

- **Resulting Behaviors**: The resulting behaviors, denoted by R, form a collection of behaviors that satisfies the environment constraints and can be produced by some implementation of the contract, depending on the contract semantics. For example, given the contract and environment constraints in Figure 6.1, the behaviors $(6, 3, 12, 6, 4)_{w,x,y,z,f}$ and $(4, 2, 8, 4, 4)_{w,x,y,z,f}$ are valid elements of R because they satisfy the environment constraints and belong to $G \cup \overline{A}$, the behavior set defining the implementation. In contrast, the behavior $(8, 4, 16, 8, 4)_{w,x,y,z,f}$ is not a valid element of R because it does not satisfy the environment constraints, and behavior $(8, 4, 8, 6, 2)_{w,x,y,z,f}$ is also invalid because it cannot be produced by any implementation of the contract under the given environment.

In this work, we differentiate between a *collection* and a *set* based on how elements are represented, even though both mathematically belong to a set of elements. A *collection* consists of explicitly enumerated elements, such as the behavior collection $\{((6, 3, 12, 6, 4)_{w,x,y,z,f}, (4, 2, 8, 4, 4)_{w,x,y,z,f}\}$. In contrast, a *set* implicitly defines its elements through formal expressions, such as first-order logic. For example, the expression $(x = 5 \wedge y = 4x)$ defines a behavior set. This distinction between collections and sets is useful in differentiating resulting behaviors from the implicitly defined sets. Simulation converts implicitly defined sets into explicitly represented collections, enabling designers to verify whether contracts align with the design intent.

6.2.2 Critical Behavior Collections and Critical Component Collections

The purpose of contract simulation is to present behaviors that allow designers to verify whether a contract aligns with the design intent. To facilitate manual verification of the contract, the behaviors should indicate whether they violate the environment, violate the guarantee, or are permitted by the specification. However, since a contract may involve numerous, or even infinite, behaviors, enumerating them all is impractical and often infeasi-

ble. Instead, techniques that help identify mistakes using a small collection of behaviors are crucial for addressing the challenge posed by the large behavior space in contracts.

Within a small collection of behaviors, some can effectively help designers identify certain errors in the contract, while others may not. For example, consider the behavior set $x \geq 3$ in the environment of a contract and two behavior collections from the set: $\{(4)_x, (3)_x, (2)_x\}$, and $\{(100)_x, (99)_x, (98)_x\}$. In the first collection, $(4)_x$ and $(3)_x$ satisfy the environment, while $(2)_x$ does not. In contrast, all behaviors in the second collection satisfy the environment. If the actual design intent is $x > 3$ or $x \geq 0.3$, the discrepancy in satisfaction within the first collection can help reveal the error. In contrast, the second collection cannot help identify the mistake, as all its behaviors still align with the designer's intent. Thus, the first collection is more informative than the second. The key difference is that the first collection contains both satisfying and violating behaviors, creating boundary crossings between the two. These boundary crossings provide opportunities to discover mistakes, particularly those related to misused operators or values in the expression. Therefore, leveraging the behavior set expression allows us to define behavior collections that are more informative for detecting mistakes.

Formally, let B be the behavior set defined by the expression. The designer's intent is represented by another behavior set D, which is not expressed through formalism. The only available operation on D is membership querying, which corresponds to presenting a behavior to the designer and asking whether it aligns with their intent.

Based on our intuition of an informative behavior collection, we define *critical behavior collections* as follows:

Definition 6.1. *A critical behavior collection R of a behavior set B is a collection that satisfies the following conditions:*

1. *Let $in(R)$ and $ex(R)$ be a partition of R such that $in(R) = R \cap B$ and $ex(R) = R \cap \overline{B}$. Both $in(R)$ and $ex(R)$ are non-empty sets.*

2. *For any pair of elements $(e_1, e_2) \in R^2$ such that $e_1 \in B$ and $e_2 \notin B$, if $e_1, e_2 \in D$ or $e_1, e_2 \notin D$, then there exists an operator in the expression that suggests a potential error at that location, assuming no other mistakes exist in the remaining part of the expression.*

The first condition ensures that R contains both elements in B and elements not in B, creating a boundary-crossing for the behavior set B. The second condition guarantees that once a discrepancy between D and B is discovered, it corresponds to a specific operator in the expression, thus helping the designer identify and correct the issue.

For example, suppose a designer's design intent is $x < 1$ but accidentally writes it as $x \leq 1$. In this case, we define the behavior set as $B = x \leq 1$ and $D = x < 1$. The behavior collection $\{(2)_x, (3)_x\}$ is not a critical behavior collection since both behaviors do not belong to B. In contrast, the behavior collection $\{(1)_x, (2)_x\}$ is a critical behavior example because

it contains $(1)_x \in B$ and $(2)_x \notin B$, which can help identify the operator $\leq$ as the potential source of the mistake.

For a given expression, multiple critical behavior collections may exist. For example, for $B = x \leq 1$, the behavior collections $\{(1)_x, (2)_x\}$, $\{(0)_x, (2)_x\}$, and $\{(0)_x, (1)_x, (2)_x\}$ are all critical behavior collections. Furthermore, in $\{(1)_x, (2)_x\}$, replacing $(2)_x$ with any other value satisfying $x > 1$ still forms a critical behavior collection. As a result, the behavior set $x > 1$ can serve as a generator for producing elements in a critical behavior example. If such a generator can be derived from the expression and a simulation procedure exists to produce behaviors accordingly, then critical behavior collections can be generated efficiently. Therefore, we define *critical component collections* as collections of behavior sets such that selecting one behavior from each set produces a critical behavior collection. The term *component* is inspired by contract literature, where it denotes a behavior set.

Definition 6.2. *A critical component collection $\mathcal{M}_{critical} = \{M_1, M_2, \ldots, M_n\}$ of a behavior set B is a collection of components (i.e., behavior sets), such that any selection of one behavior from each component forms a critical behavior collection of B. Formally,*

$$R = \{e_1, e_2, \ldots, e_n\}, \text{where } e_i \in M_i \quad \forall i \in \{1, \ldots, n\}, \tag{6.1}$$

must be a critical behavior collection. The subset $in(\mathcal{M}_{critical})$ consists of components that produce $in(R)$, while $ex(\mathcal{M}_{critical})$ consists of components that produce $ex(R)$.

For example, $\mathcal{M}_{critical} = \{(x > 1), (x = 1), (x < 1)\}$ is a critical component collection for $x \geq 1$, with $in(\mathcal{M}_{critical}) = \{(x > 1), (x = 1)\}$ and $ex(\mathcal{M}_{critical}) = \{(x < 1)\}$.

6.2.3 Methodology Overview

Combining the purpose of simulation with the notions of critical behavior collections and critical component collections, we propose a simulation methodology that: 1. Produces behaviors that satisfy the given constraints. 2. Automatically generates critical behavior collections to assist designers in verifying whether a written contract aligns with the design intent.

As shown in Figure 6.2, our proposed contract simulation methodology takes a contract and environment constraints as input. *Environment Generation* and *Implementation Generation* produce critical component collections from contract expressions to facilitate the verification of both contract environments and implementations. The components generated from environment generation consist of collections of environments, whose behaviors help the designer verify whether the environment is correctly defined. Similarly, the components generated from implementation generation consist of collections of implementations, whose behaviors assist in verifying whether the implementation aligns with the expected design intent.

To produce the collection of behaviors as resulting behaviors, we develop *Constraint-based Simulation*. The simulation produces behaviors that satisfy constraints from inputs, a

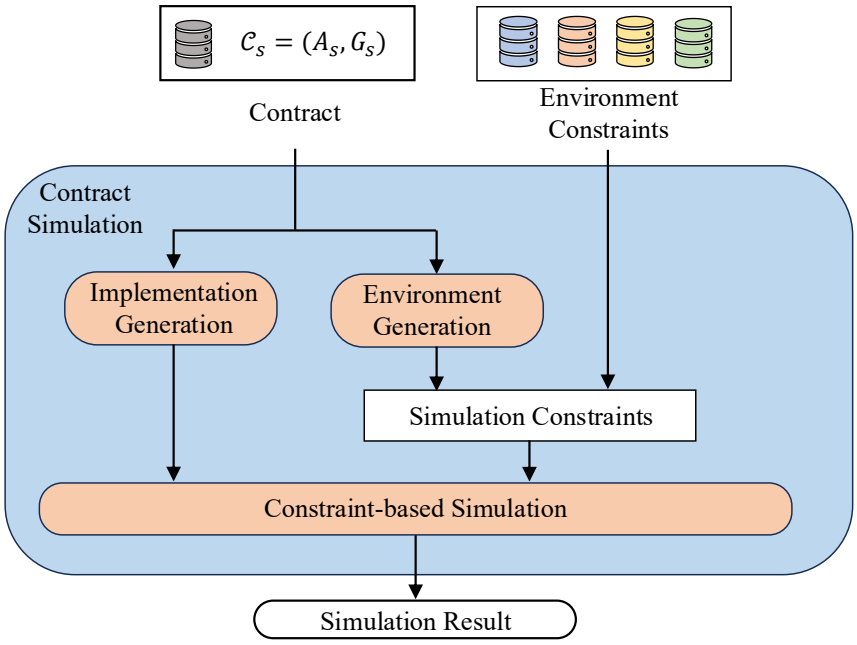


Figure 6.2: Overview of the proposed contract simulation methodology.

generated environment, and a generated implementation. It enables the automatic generation of critical behavior collections from the critical component collections. The constraints combine the generated environment with the environment constraints to form the *Simulation Constraints*. These constraints guide the simulator in producing critical behavior collections for both environments and implementations, allowing the designer to review them.

6.3 Automated Component Generation

As introduced, critical component collections can guide the simulation to produce critical behavior collections. Generating critical component collections from expressions in contracts is therefore a crucial step to ensure coverage of potential mistakes, such as misused operators and incorrect values, in contracts. To achieve this, this section presents an *Automated Component Generation* algorithm, which derives critical component collections from contract expressions.

6.3.1 Algorithm

In the examples of critical behavior collections and critical component collections, the operator and its operands (i.e., the values in the example) define the *boundary* of the behavior set.

This boundary is determined by the function represented by the operator and its operands. The operand values that evaluate the function to true fall on one side of the boundary. As a result, critical component collections can be formed by categorizing operand values based on their effect on the function's output. For instance, consider the behavior set expression $B = (x \leq 1)$. Any value of x satisfying $x \leq 1$ results in a true evaluation, ensuring that behaviors with such values of x belong to B. Conversely, values of x satisfying $x > 1$ yield a false evaluation, indicating that behaviors with such values of x do not belong to B. By comparing the operands x to 1, we can construct a critical component collection $\mathcal{M}_{critical} = \{(x < 1), (x = 1), (x > 1)\}$. Each component in this collection corresponds to a distinct condition that determines whether the function evaluates to true or false, helping identify potential errors in the contract's definition. Therefore, the values of operands and their relationships can be leveraged to systematically guide the construction of critical component collections.

When an expression contains multiple operators, the effect of a specific operand on the function output may be offset by values from other parts of the expression, making it difficult to ensure that the constructed components form a valid critical component collection. Therefore, isolating the effects of other parts of the expression is crucial to ensure that the influence of operand values and their relationships propagates to the evaluation result of the entire expression. Given a target operator for constructing a critical component collection, the values of other parts of the expression must be controlled so that the resulting value of the operator connecting them depends on the output of the target operator.

For example, consider the behavior set expression $B = (x < 10) \wedge b$, where x is a real-valued variable, and b is a boolean variable. The expression contains two operators: $\wedge$ and $<$. For the operator $<$, the operands of the targeted operator $(x < 10)$ can be utilized to form a critical component collection $\{(x < 10), (x = 10), (x > 10)\}$. However, when b is false, the evaluation of the entire expression results in: $\{(x < 10) \wedge false, (x = 10) \wedge false, (x > 10) \wedge false\}$. Since all components evaluate to false regardless of the value of x, the collection $\{(x < 10) \wedge b, (x = 10) \wedge b, (x > 10) \wedge b\}$ is not a critical component collection. To ensure that the effect from the evaluation of $(x < 10)$ propagates to the evaluation result of the entire expression, we need to control b so that it does not override the output of the target operator $<$. Specifically, setting $b = \text{true}$ ensures that the evaluation of $x < 10$ influences the overall expression. Consequently, the resulting critical component collection is $\mathcal{M}_{critical} = \{((x < 10) \wedge (b = \text{true})), ((x = 10) \wedge (b = \text{true})), ((x > 10) \wedge (b = \text{true}))\}$. The purpose here is to guarantee that the resulting components form a valid critical component collection. This means that the components must not all belong to B or all outside of B, preserving the boundary-crossing property for detecting discrepancies. If such isolation cannot be achieved, the targeted operator becomes redundant since its evaluation never influences the overall expression.

In this way, a critical component collection can be constructed for each operator. Once the critical behavior collection from each behavior set is generated, the designer can verify each operator by examining whether the relationship between the generated behaviors and B aligns with the intended design.

Operator p and the Expression Without Other Parts	$in(\mathcal{M}_{critical})$	$ex(\mathcal{M}_{critical})$	Isolation Rule (Targeted Part: Isolation Result)
$\phi_A \wedge \phi_B$	$\phi_A \wedge \neg\phi_B$	$\phi_A \wedge \neg\phi_B$ $\neg\phi_A \wedge \phi_B$ $\neg\phi_A \wedge \neg\phi_B$	$\phi_A : \phi_A \wedge \phi_B$ $\phi_B : \phi_A \wedge \phi_B$
$\phi_A \vee \phi_B$	$\phi_A \wedge \phi_B$ $\neg\phi_A \wedge \phi_B$ $\phi_A \wedge \neg\phi_B$	$\neg\phi_A \wedge \neg\phi_B$	$\phi_A : \phi_A \wedge \neg\phi_B$ $\phi_B : \neg\phi_A \wedge \phi_B$
$\phi_A \implies \phi_B$	$\phi_A \wedge \phi_B$ $\neg\phi_A \wedge \phi_B$ $\neg\phi_A \wedge \phi_B$	$\phi_A \wedge \neg\phi_B$	$\phi_A : \phi_A \wedge \neg\phi_B$ $\phi_B : \phi_A \wedge \phi_B$
$\neg(\phi_A)$	$\neg\phi_A$	ϕ_A	$\phi_A : \phi_A$
$e_1 \leq e_2$	$e_1 < e_2$ $e_1 = e_2$	$e_1 > e_2$	
$e_1 \geq e_2$	$e_1 > e_2$ $e_1 = e_2$	$e_1 < e_2$	
$e_1 < e_2$	$e_1 < e_2$	$e_1 > e_2$ $e_1 = e_2$	
$e_1 > e_2$	$e_1 > e_2$	$e_1 < e_2$ $e_1 = e_2$	
$e_1 = e_2$	$e_1 = e_2$	$e_1 \neq e_2$	
$e_1 \neq e_2$	$e_1 \neq e_2$	$e_1 = e_2$	

Table 6.1: Examples of rules for constructing critical component collections and applying isolation to ensure the effect of a single operand can influence the evaluation outcome.

Algorithm 5 Automated Component Generation

Inputs: A behavior set represented by the expression ϕ
Output: Critical component collections C where each operator corresponds to one collection.
1: $C \leftarrow \{\}$
2: $p \leftarrow$ `getMainOperator`(ϕ)
3: `automaticComponentGenerationTraversal`(ϕ, C, p)
4: **return** C

Algorithm 6 `automaticComponentGenerationTraversal`

Inputs: A behavior set represented by the expression ϕ, current collection C, target operator p
1: **for all** operands ϕ_{sub} of p **do**
2: $p_{sub} \leftarrow$ `getMainOperator`(ϕ_{sub})
3: $\phi \leftarrow$ `isolateUnrelatedPart`(ϕ_{sub}, ϕ)
4: `automaticComponentGenerationTraversal`(ϕ, C, p_{sub})
5: $\phi \leftarrow$ `isolationRecovery`(ϕ_{sub}, ϕ)
6: $C[p] \leftarrow$ `ApplyRule`(p, ϕ)
7: **return**

Based on this concept, we develop an algorithm to generate a critical component collection for each operator. Assuming that an expression is represented as a syntax tree, illustrated

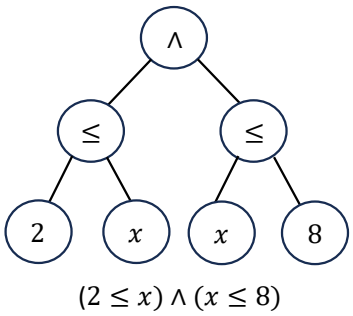


Figure 6.3: Example of the syntax tree for the expression $(2 \leq x) \wedge (x \leq 8)$.

in Figure 6.3, where operators, constant values, and variables form the nodes, and edges represent operand connections, the algorithm constructs critical component collections by traversing the syntax tree. As outlined in Algorithms 5 and 6, the algorithm employs a depth-first search (DFS) traversal of the syntax tree to generate critical component collection for each operator. This approach ensures that the isolation of a target operator can be effectively reused for all operators within its subexpression, leading to improved computational efficiency in the generation process. Algorithm 5 initializes the resulting collections and then invokes the traversal process defined in Algorithm 6 on the operator at the root of the syntax tree to generate critical components for each operator in the expression. In Algorithm 6, Line 3 modifies the expression to ensure the isolation of the effect from other parts of the expression. After exploring all operators in a subtree, Line 5 restores the expression to allow traversal of the next subtree. Lines 1–5 perform a DFS traversal of the syntax tree. Once critical component collections for all operators in the subtrees are created, Line 6 applies rules to construct a critical component collection for the target operator, leveraging operand values and their relationships. Table 6.1 shows examples of rules for constructing critical component collections and isolating effects, considering operators such as $\wedge$, $\vee$, $\implies$, $\neg$, $=$, $\neq$, $\geq$, $>$, $\leq$, and $<$.

6.3.2 Examples

Consider the contract $\mathcal{C} = (2 \leq x \leq 8, z = 2x \wedge y = 2w)$. For clarity, we label each operator with an index to distinguish them, resulting in: $(2 \leq_1 x \wedge_2 x \leq_3 8)$ and $(z =_4 2x \wedge_5 y =_6 2w)$. The assumption, $2 \leq_1 x \wedge_2 x \leq_3 8$, is passed to the automated component generation algorithm for environment generation.

The root operator in the assumption is $\wedge_2$, with $\phi_A = (2 \leq_1 x)$ and $\phi_B = (x \leq_3 8)$ as its operands. First, the operand ϕ_A is considered, and the isolation rule for $\wedge$ is applied, leading to $\phi_A \wedge \phi_B = 2 \leq_1 x \wedge x \leq_3 8$. This ensures that the truth value of $2 \leq_1 x$ can affect the evaluation of the entire expression. Then, the operand $\leq_1$ of ϕ_A is considered. Since there are no further expressions or operators as subtrees in its syntax tree, no additional

traversal or isolation is needed. Consequently, the rule for constructing critical component collections is applied to ϕ_A and combined with the isolation part, leading to the following critical component collection:

$$C[\leq_1] = \{(2 < x \wedge x \leq 8), (2 = x \wedge x \leq 8), (2 > x \wedge x \leq 8)\}.$$

After completing the $2 \leq_1 x$ part, the isolation is recovered.

Next, the operand ϕ_B is considered. Similar to ϕ_A, this leads to the following critical component collection:

$$C[\leq_3] = \{(2 \leq x \wedge x < 8), (2 \leq x \wedge x = 8), (2 \leq x \wedge x > 8)\}.$$

Finally, the algorithm completes the traversal of the root operator $\wedge_2$ by constructing its critical component collection:

$$C[\wedge_2] = \{(2 \leq x \wedge x \leq 8), (2 \leq x \wedge \neg(x \leq 8)), (\neg(2 \leq x) \wedge (x \leq 8)), (\neg(2 \leq x) \wedge \neg(x \leq 8))\}.$$

We can observe that $C[\leq_1]$ contains the components $(2 < x \leq 8)$, $(x = 2)$, and $(x < 2)$ for determining if the expression of $\leq_1$ is correct. Similarly, $C[\leq_3]$ contains the components $(2 \leq x < 8)$, $(x = 8)$, and $(x > 8)$, while $C[\wedge_2]$ contain the components $(2 \leq x \wedge x \leq 8)$, $(x > 8)$, and $(x < 2)$. These collections allow the designer to verify the correctness of the contract's expressions by checking if the behaviors align with the design intent. Each component represents a boundary condition that reflects how the operands and operators interact within the overall expression.

Similarly, for the implementations derived from the contract, the critical component collections can be generated as follows:

$$\begin{aligned}
C[=_4] &= \{(z = 2x \wedge y = 2w), (z \neq 2x \wedge y = 2w)\},\\
C[=_6] &= \{(z = 2x \wedge y = 2w), (z = 2x \wedge y \neq 2w)\},\\
C[\wedge_5] &= \{(z = 2x \wedge y = 2w), (z = 2x \wedge \neg(y = 2w)),\\
&\qquad (\neg(z = 2x) \wedge y = 2w), (\neg(z = 2x) \wedge \neg(y = 2w))\}.
\end{aligned}$$

These collections of critical components are then used in the constraint-based simulation to generate the corresponding critical behavior collections.

6.3.3 Analysis

Here we analyze the complexity of the algorithm. Consider an expression with n operators. Since the algorithm follows a DFS approach, its complexity is $O(|V| + |E|)$, where $|V|$ is the number of vertices in the graph, and $|E|$ is the number of edges. As a syntax tree is a tree, it contains $|V| = n$ vertices, and $|E| = n - 1$ edges. Thus, the complexity is $O(|V|+|E|) = O(n)$. This ensures that the algorithm can efficiently handle expressions with a large number of operators.

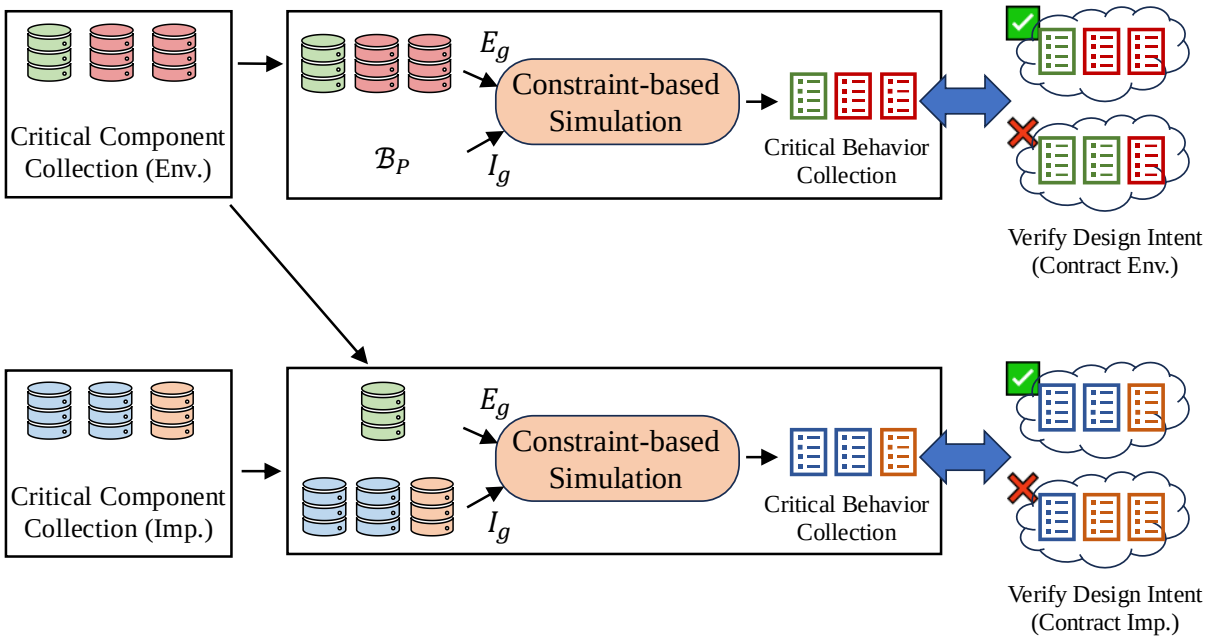


Figure 6.4: Illustration of using constraint-based simulation to generate critical behavior collections from generated environments and implementations to verify design intent. Different colors indicate the satisfaction or violation of the contract environment or contract implementations.

Note that if we apply isolation rules separately, rather than using our DFS approach, the number of steps for isolation depends on the height of each operator in the tree (the distance from the root). In the worst-case scenario, where the tree is unbalanced, the number of isolation steps becomes $\sum_{i=1}^{n} i = O(n^2)$, which results in higher computational complexity than our proposed method.

6.4 Constraint-based Simulation

Critical component collections generated from environments and implementations need to be converted into critical behavior collections for designers to review behaviors and verify design intent. Constraint-based simulation facilitates this by generating behaviors that adhere to contract semantics while following the guidance of the generated components. For each generated environment E_g, the simulation produces a behavior from its behavior set. The behaviors from a critical component collection form a critical behavior collection for the contract environment. For a generated implementation I_g, the simulation constructs simulation constraints by combining a generated environment E_g with the environment constraints and then generates behaviors that satisfy both contract semantics and these constraints.

Figure 6.4 illustrates the role of constraint-based simulation in the process. The designer compares the reported satisfaction and violation behaviors with the design intent to ensure the contract is correctly written.

This section presents an SMT-based algorithm for generating behaviors from the given environment and implementation. Note that constraint-based simulation operates independently of automatic component generation. The environment and implementation can be any contract component, such as an entire assumption or guarantee behavior set. This flexibility allows the simulation to accommodate various use cases and different strategies for generating critical component collections, such as sweeping input values to observe corresponding changes in output values.

6.4.1 Algorithms

To generate behaviors allowed or disallowed by contracts, contract semantics must be considered. Although the generated implementations include both implementations that satisfy the contract and one that violates it, the resulting behaviors also depend on whether the environment meets the specified contract environments. The constraints provided for simulation, referred to as the simulation constraints, consists of an environment and optional environment constraints. The environment constraints allow designers to guide the simulation by focusing on specific subsets of environments. If the simulation constraints satisfy the contract environment, the resulting behaviors align with those of the implementations. Conversely, if the simulation constraints violate the contract environment, the resulting behaviors are not required to adhere to the implementations, as the specification permits the system to produce any behaviors when operating outside the specified environment.

Therefore, the behaviors permitted by the contract depend on whether the simulation constraints satisfy the environment. We refer to the behaviors under a given set of simulation constraints as *promise behaviors*, which represent behaviors determined after considering both simulation constraints and contract semantics.

The simulation problem aims to generate behaviors from the promise behaviors as the resulting behaviors. These promise behaviors are defined by expressions describing the environment constraint, environment, and implementation. Since a behavior corresponds to a value assignment of variables in the expression, the problem can be transformed into an SMT problem, which determines whether such an assignment exists. As long as the required background theory for the expression is supported by an SMT solver, the solver can return values representing behaviors whenever the promise behavior set is nonempty. These returned values constitute the resulting behaviors of the simulation.

Based on this concept, we propose *Constraint-based Simulation*, outlined in Algorithm 7. Lines 1–5 check whether the simulation constraints satisfy the environment and determine the promise behaviors $B_{promise}$. Lines 7-12 formulate the SMT problem by operating on the set represented by the expressions from the simulation constraints and the implementation, and leverage an SMT solver to generate a resulting behavior. The process loops to generate distinct behaviors, up to the requested number n. Once a behavior is generated, Lines 8 and

Algorithm 7 Constraint-based Simulation

Inputs: A contract $\mathcal{C} = (\mathcal{E}, \mathcal{I})$, a generated implementation $I_g \in \mathcal{I}$, simulation constraints $E_s = E_c \cap E_g$, number of distinct behaviors n.

Output: A collection of at most n distinct behaviors that meet the contract semantics following the environment E_s and implementationI_g.

```
1: // check the environment
2: if E_s ∉ E then
3:     B_promise ← E_s
4: else
5:     B_promise ← I_g ∩ E_s
6: R ← []
7: for i = 1 → n do
8:     B_promise ← B_promise − R
9:     r_sat, b ← SMT_generate(B_promise)
10:    if not r_sat then
11:        return R, i − 1
12:    R ← Append(R, b)
13: return R, n
```

12 exclude the behavior from the promise behavior set. In first-order logic, the exclusion is achieved by introducing a constraint that removes the generated behavior from $B_{promise}$. For example, if the promise behaviors are defined by $5x \leq y \leq 7x$ and a behavior $(2, 10)_{x,y}$ is produced, we introduce the constraint $\neg(x = 2 \wedge y = 10)$, resulting in the following new promise behavior set:

$$\begin{aligned} &(5x \leq y \leq 7x) \wedge \neg(x = 2 \wedge y = 10) \\ =\ &(5x \leq y \leq 7x) \wedge (x \neq 2 \vee y \neq 10), \end{aligned}$$

which excludes the previously generated behavior $(2, 10)_{x,y}$. The algorithm terminates when either the requested number n of behaviors is reached or the SMT solver returns unsat, indicating that no additional distinct behaviors exist under the given constraints.

The algorithm may return an empty collection if the SMT problem is unsatisfiable in the first iteration, indicating that the promise behavior set is empty. This can occur due to:

1. Conflicting constraints in the simulation setup, leading to $E_s = \emptyset$.
2. An empty implementation due to conflicting expressions during automatic component generation.
3. No violating behaviors when the implementation represents a disallowed behavior under the contract.
4. Non-receptive contracts [182], where the contract has no behaviors under certain environments.
5. Over-constrained conditions, where the simulation environment controls the output.

These cases can be identified by examining E_s, I_g, and the contract specifications.

6.4.2 Example

We illustrate the constraint-based simulation using the example in Figure 6.1. Consider the contract $\mathcal{C} = (2 \leq x \leq 8, z = 2x \wedge y = 2w)$ and the environment constraint $E_c = (2 \leq x \leq 3 \wedge w = z \wedge f = y/x)$. Assume we aim to generate two behaviors from the environment $E_g = (2 \leq x \leq 8)$ and the implementation $I_g = (z = 2x \wedge y = 2w)$. The simulation constraints are then computed as $E_s = (2 \leq x \leq 8) \wedge (2 \leq x \leq 3 \wedge w = z \wedge f = y/x) = (2 \leq x \leq 3 \wedge w = z \wedge f = y/x)$.

Step 1: Determine the Promise Behaviors: Since the simulation constraints $(2 \leq x \wedge 3 \wedge w = y \wedge f = y/x)$ always satisfy the contract environment, the promise behavior set consists of all behaviors allowed by the implementation under the simulation constraints:

$$\begin{aligned} B_{promise} &= E_s \cap I_g \\ &= (z = 2x \wedge y = 2w) \wedge (2 \leq x \leq 3 \wedge w = z \wedge f = y/x). \end{aligned}$$

Step 2: Generate the First Behavior: The promise behavior $B_{promise}$ is formulated as an SMT problem, where any satisfying assignment represents a valid resulting behavior. Using an SMT solver, we can obtain a behavior as a satisfying assignment. Assume the solver returns $(4, 2, 8, 4, 4)_{w,x,y,z,f}$, which is stored in the result collection R as the first behavior.

Step 3: Generate the Second Behavior: To ensure uniqueness, we add the following constraint to the promise behaviors to exclude the first behavior:

$$(w \neq 4 \vee x \neq 2 \vee y \neq 8 \vee z \neq 4 \vee f \neq 4).$$

The updated promise behavior set defines a new SMT problem. Solving it again, assume the solver returns $(6, 3, 12, 6, 4)_{w,x,y,z,f}$, which is added to R as the second behavior.

Finally, since we have reached the requested count of two behaviors, the algorithm terminates. The resulting collection of behaviors is

$$R = \{(4, 2, 8, 4, 4)_{w,x,y,z,f}, (6, 3, 12, 6, 4)_{w,x,y,z,f}\}.$$

6.4.3 Complexity Analysis

The complexity of the algorithm depends on the SMT solver, which varies based on the background theories required by the expressions for sets. To illustrate the complexity in terms of the number of SMT variables and clauses, we assume a first-order logic formalism and assume-guarantee contracts.

Let the system contain m ports, including evaluation ports encoded in the environment constraints. The sizes of the assumption, generated environment, generated implementation, and simulation constraints are defined as s_A, s_{ge}, s_{gi}, and s_{sc} respectively, where size is measured by the number of literals and operations used to encode the set. For environment checking, the SMT instance size is $O(s_A + s_{ge} + s_{sc})$ and m variables. For generating the ith behavior, the SMT instance size is $O(s_{sc} + s_{gi} + m \times i)$ with m variables, where $m \times i$ accounts for the additional constraints to exclude previously generated behaviors.

6.5 Experiments

To assess the effectiveness and efficiency of the simulation methodology, we implemented the simulation methodology in Python, utilizing Z3 [48] as the SMT solver. The evaluation was conducted on an Intel I9-9980HK machine with 32GB of memory.

Since there is no prior work in contract simulation, we conducted three experiments to demonstrate that the proposed methodology and algorithms can effectively assist designers in verifying design intents. The first experiment verifies the correctness of the simulation results by sweeping through different input values using the example shown in Figure 6.1. The second experiment examines the scalability of the automatic component generation algorithm to demonstrate that the algorithm is sufficiently efficient for generating environments and implementations. The third experiment focuses on the scalability of the constraint-based simulation algorithm, showcasing that the simulation algorithm is efficient in producing behaviors as critical behavior collections for verification. In the following, we will elaborate on the experiment settings and their results.

6.5.1 Input Sweeping

This experiment sweeps different input values for the contract shown in Figure 6.1 to examine if the resulting behaviors align with the contract semantics and the environment constraints. The contract is defined as $\mathcal{C} = (2 \leq x \leq 8 \wedge z = 2x \wedge y = 2w)$, subject to the connection $w = z$ and objective $f = y/x$, similar to the example.

To sweep the inputs for different values of x, the constraint $x = x_{input}$ is added for each simulation, , where x_{input} is a number between $[2, 8]$. Therefore, the environment constraint for each value of x_{input} is $E_c = (x = x_{input} \wedge w = z \wedge f = y/x)$. The value of x_{input} is swept from 2 to 8, with a step size of 0.2, to observe the relationship between y and f for different input values of x. In this case, the number of possible behaviors is exactly one, as the constraints and the connections define unique output values $y = 4x$ and $f = 4$ for every input value. This ensures that for each chosen x_{input}, the corresponding values of y and f are predictable, making it easier to verify the correctness of the results.

Figure 6.5 shows the simulation results. The resulting value of y equals $4x_{input}$, as demonstrated by the line, which is the expected value according to the contract. For the value of f, the resulting value is a constant 4, which also matches the expected value. These results demonstrate that our constraint-based simulator can produce behaviors based on contracts and environment constraints. This capability allows designers to examine the proposed contract, observe how the specification defines the system in different environments, and verify whether the results match the design intent.

6.5.2 Scalability of Automated Component Generation

Next, the experiment on the scalability of the automated component generation algorithm aims to demonstrate the efficiency of the proposed algorithm in generating critical component

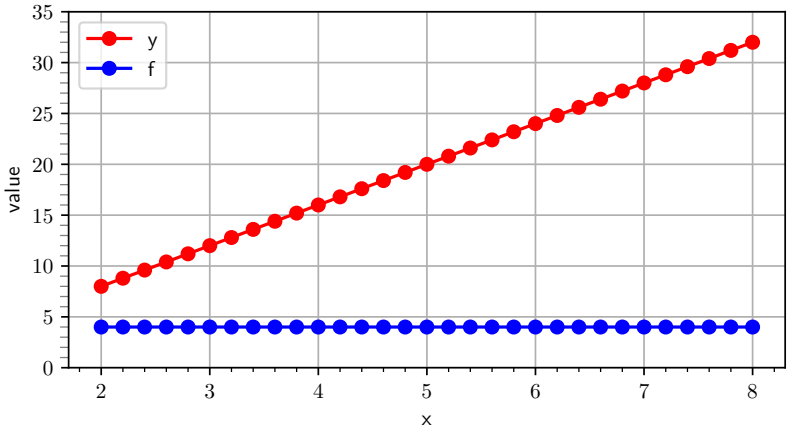


Figure 6.5: Values of y and f according to different values of x.

d \ 2^n	2^1	2^2	2^3	2^4	2^5	2^6	2^7	2^8	2^9	2^{10}
# Critical Component Collections	3	7	15	31	63	127	255	511	1023	2047
Time w/ copy(s)	0.0011	0.0054	0.0236	0.0891	0.4441	2.2256	9.7702	42.9984	198.79	1006.60
Time w/o copy(s)	1.777E-5	3.667E-05	7.992E-5	1.549E-4	0.0003	0.0006	0.0012	0.0023	0.0051	0.0090

Table 6.2: The execution time of the automatic component generation algorithm and the number of components generated under different input sizes.

collections. We adopted first-order logic and polynomial arithmetic as the background theory.

To define the size of the input contracts, we define an expression as a variable, a constant, or arithmetic operations $(+, -, \times, \div)$ on variables and constants. A clause is created by connecting expressions with a comparison operator: $(\leq, <, \geq, >, =, \neq)$. The input formula is a first-order formula constructed by connecting clauses using logical operators $(\wedge, \vee, \neg)$ and parentheses. Thus, the formula can be represented as a syntax tree, where clauses are the leaf nodes, and logical operators serve as non-leaf nodes, structuring the formula into a hierarchical form.

Given a tree depth of n, the input size d is defined as 2^n, representing the total number of clauses in the tree. In our experiments, without loss of generality, we consider input formulas whose syntax trees are complete binary trees, ignoring the negation operator for simplicity. The syntax trees are randomly generated using 100 variables, ensuring that they maintain a complete binary tree structure.

The experiment compares the execution time of the automated component generation algorithm against the number of clauses 2^n, for different values of n ranging from 1 to 9.

The results, as summarized in Table 6.2, show that the execution time grows exponentially with increasing input size, approximately quadrupling when the input size doubles, despite the algorithm's linear complexity. This discrepancy arises because the number of generated

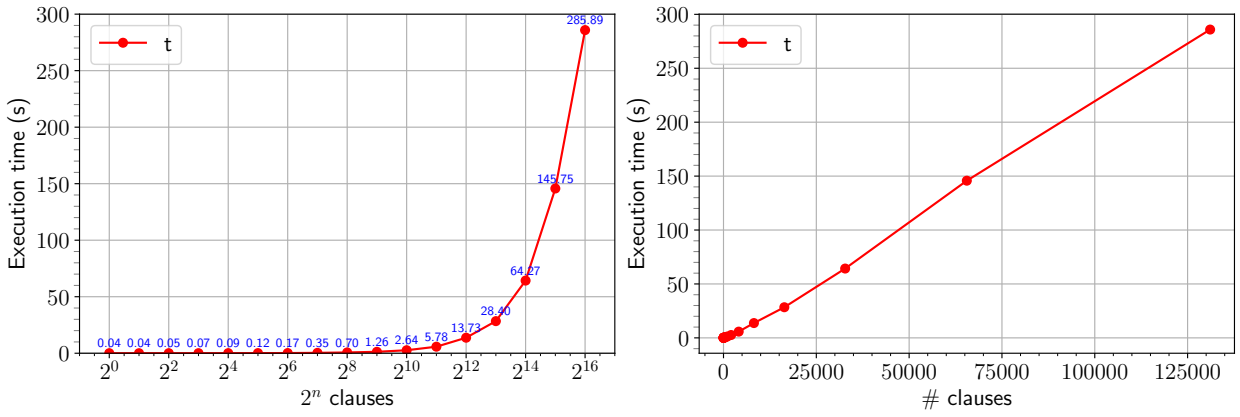


Figure 6.6: The execution times of the constraint-based algorithm under different numbers of clauses 2^n.

components scales with the number of clauses 2^n and the copying each formula incurs an additional 2^n factor, leading to an overall growth of $2^n \times 2^n = 4^n$ as n increases, explaining the observed quadruple increase.

This overhead can be mitigated by avoiding unnecessary formula copying. Automated component generation can modify the formula in place and produce components as they are found. The results in the last row of Table 6.2 demonstrate that with this optimization, execution time grows linearly with the number of clauses, confirming the algorithm's linear complexity and its efficiency in generating all critical component collections for a given formula.

In practice, designers may neither need nor be able to verify the generated behaviors when dealing with a large number of operators. From the perspective of contract-based design methodology, human-written contracts should be structured into multiple scenarios and viewpoints to improve manageability and better reflect the design intent. Furthermore, verifying design intent using critical behavior collections can only be performed by the designer who holds the design intent. Manually examining tens of thousands of behaviors is both impractical and inefficient. For instance, if verifying a single behavior takes 10 seconds, checking 10,000 behaviors would require approximately 28 hours—more than three full workdays. Additionally, more complex formulas may require even longer verification times, making exhaustive manual inspection infeasible. As a result, the ability to handle contract sizes up to 2^{10} is sufficient to support the purpose of checking design intent.

6.5.3 Scalability with Behaviors

In this experiment, the scalability of the constraint-based simulation algorithm is evaluated based on the size of the input contracts. The contract size is defined as the number of clauses in the formulas representing the environment and implementation. For consistency, the same

syntax tree representation and random generation procedure from the previous experiment are adopted. The contract formalisms used are assume-guarantee contracts, where both the environment and implementation are expressed as first-order logic formulas represented by syntax trees. The assumptions include 10 variables, while the guarantees introduce an additional 90 variables, resulting in a total of 100 variables per contract. The numbers of clauses of environment and implementation are set to the same number of clauses 2^n.

The experiment measures execution time as a function of the number of clauses, 2^n, with n ranging from 0 to 15. The simulation constraints are derived from a generated environment to ensure that the constraints handling are included in the algorithm while guaranteeing that the environment is valid for the contract. To focus on the scalability of constraint-based simulation, the execution time measurement excludes the time required to generate the environment.

The result, summarized in Figure 6.6, shows that the execution time increases linearly with the number of clauses, as indicated by the linear trend in the execution time. The results demonstrate that the algorithm can efficiently handle contracts with tens of thousands of clauses (approximately 2^{15}), allowing designers to obtain results within a few minutes. As discussed in previous experiments, designers should leverage contract-based design to structure contracts with separate conditions and viewpoints. Since the simulation does not need to handle extremely large contract sizes in practice, the observed execution time is reasonable, confirming that the efficiency of the proposed algorithm is practical for verifying design intent.

Note that other background theories and formulas for SMT solving may result in different complexity, This experiment shows that contracts formulated with first-order contracts can be efficiently simulated.

6.6 Conclusion

This chapter presented a simulation methodology for contract-based design, which allows designers to observe specified system behaviors and verify that their design intent aligns with the written contracts. The methodology integrates automatic component generation with constraint-based simulation. Automatic component generation separates the environment and implementation sets in a contract, helping designers identify potential errors in contract expressions and facilitate their correction by the proposed concept of critical behavior collections and critical component collections. Constraint-based simulation uses the environment constraints and the generated components to produce collections of behaviors for review. These constraints not only support the evaluation of specific input conditions but also help analyze system behaviors under various connections and evaluation metrics. Experimental results demonstrated that the proposed algorithm provides correct simulation results according to contract semantics and constraints. Additionally, it effectively generates critical behavior collections with reasonable runtime, making it a practical tool for contract-based system development and management.

As the first work in contract simulation, this chapter opens up new research opportunities and methodologies for applying contract-based design. Future work includes (1) enhancing automatic generation to provide greater flexibility in controlling the environment and implementation generation, (2) applying the methodology to different set expressions, such as linear temporal logic, and (3) the development of tools that leverage this methodology for various CPS applications.

Chapter 7

Synthesis: Component Selection using Behaviors

Contract-based component selection is one of the crucial problems in automated contract synthesis, which reduces design time and cost by encouraging the reuse of subsystem designs from an existing library. However, existing techniques assume the objective function is expressed solely with component parameters, such as size, cost, and power consumed, The assumptions imposes the burden of characterizing components with parameters and deriving the appropriate objective as a function of these parameters, overlooking behavior abstractions that could make the selection process more effective. This chapter proposes a contract-based component selection algorithm that consists of two parts: a contract-based system reasoning part that guides the selection and a black-box optimizer that selects the final choice. The contract-based system-reasoning part can evaluate, verify, and suggest the selection based on system behavior using contract operations to guide the black-box optimizer. Experimental results based on the design problem for an unmanned aerial vehicle propulsion system, show that the proposed methods can successfully find and optimize component selection for all test cases within the time limit and outperform the existing methods.

7.1 Introduction

As the scale of cyber-physical systems grows, design complexity and heterogeneity result in prolonged design cycles and high costs to fulfill the design requirements and optimize performance. Various methodologies have been proposed to address heterogeneity and complexity [158]. Among them, contract-based design [119, 151] is a promising design methodology that integrates formal specifications in the general framework of platform-based design, enabling early virtual integration tests and decomposition of the problem for large system design [43, 121, 164].

Component selection is a critical step in the platform-based design methodology to ensure that the system meets the requirements and to optimize the system's performance and

costs. This step involves mapping subsystems to available options in a library to meet design specifications and optimize the objective function that estimates the performance. Figure 7.1 (a) illustrates an unmanned air vehicle (UAV) component selection problem that involves choosing motors, batteries, and propellers to meet design requirements and optimize objectives based on system behaviors. System behavior refers to the system's outputs in response to environmental inputs, which can be from a static perspective or dynamically with temporal information. For instance, a UAV behavior consists of values such as voltage, current, power, thrust, and position. Component selection methods without contracts demand extensive efforts to model target systems rigorously, requiring the formulation of a mathematical programming problem for each particular design problem [55, 88, 89]. In contrast, the contract-based component selection process relieves designers from adjusting the mathematical programming problem when the design problem is modified.

Various algorithms to meet design specifications and/or optimize performance using contract-based component selection have been proposed. Peter *et al.* introduced a satisfiability modulo theories (SMT)-based component-based synthesis that encodes the selection of components into SMT formulas to satisfy the system properties [133]. Mishra and Jagannathan proposed a bi-directional specification-guided synthesis procedure with conflict-driven learning for components library specified in Hoare-style pre- and post-conditions [113]. These two approaches focus on generating a valid selection that meets the design specification without optimization considerations in the selection. However, in CPS design problems, the selection may profoundly impact the design performance and cost, necessitating an optimization across all valid selections.

To include optimization over the selection, Iannopollo *et al.* proposed a counter-example-guided inductive synthesis (CEGIS)-based constrained synthesis flow to compose and select contracts simultaneously from a library of components specified in linear temporal logic (LTL) contracts [73]. In a subsequent paper, they improved the flow by decomposing the contracts [75] to reduce complexity. Oh *et al.* presented a parameter-based synthesis that explores the parameters of contracts using bi-level optimization to minimize the cost function while ensuring robustness [129]. However, these approaches rely on oversimplified assumptions, i.e., that system behavior and cost can be abstracted as component parameter values, and that the objective function can be expressed as a summation of terms, each expressed by the parameters of a component. These assumptions lead to limitations and disadvantages. First, parameters represent an abstraction of component behaviors and the abstraction may not apply to all available components for selection, For example, a bipolar junction transistor (BJT) has a different set of parameters and characteristics from a metal–oxide–semiconductor field-effect transistor (MOSFET) [155]. Using parameters requires designers to make additional abstractions and derive an objective function from component parameters, which could be challenging or even infeasible. On the contrary, treating parameters as ports in a system with static behaviors allows us to formulate the selection objective using behaviors. Additionally, the assumption of summation of terms ranks the components based solely on their parameters, neglecting the compatibility between components in terms of behaviors. Each component might perform well with a different set of components. Consequently, this

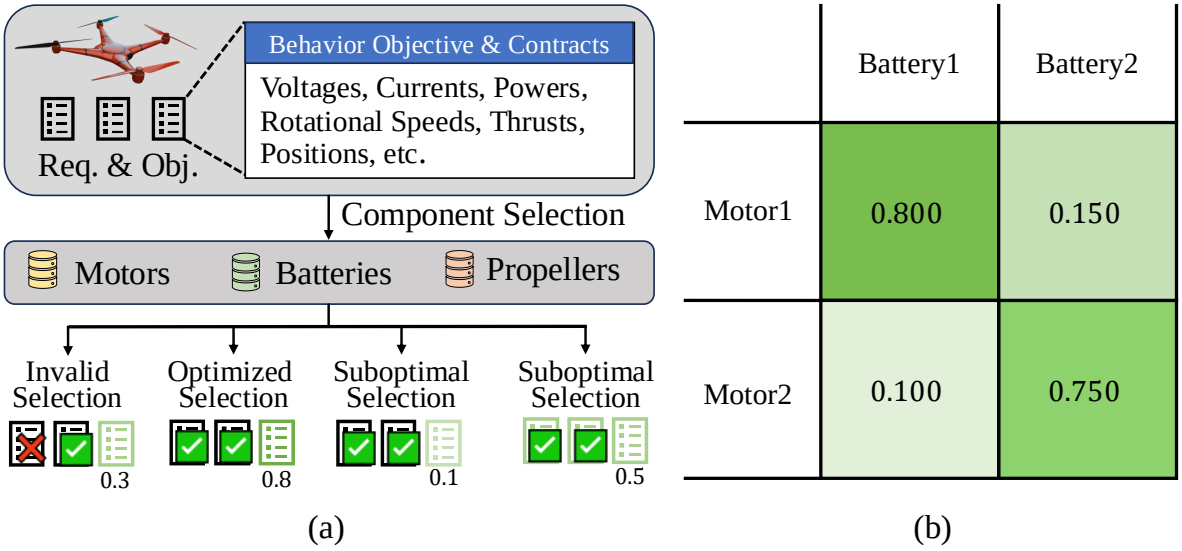


Figure 7.1: (a) An example UAV component selection problem using behavior as its design objectives and requirements. (b) An example scenario that is challenging for parameter-based optimization.

assumption could result in a suboptimal selection. In Figure 7.1 (b), the best selection and the second best selection are disjoint, suggesting that the assumption does not apply to this case. While Oh *et al.* [129] did not explicitly state the assumption for the objective function, the case study and the absence of details for objective function evaluation suggest the same assumption.

To remedy these drawbacks, we propose a contract-based component selection algorithm defined by behaviors instead of component parameters. Our contributions are summarized as follows:

- The proposed algorithm can handle constraints and objectives expressed by the available behaviors of the selection. To the best of our knowledge, this is the *first* work that considers an objective function using behaviors.
- We propose a contract-based system reasoning algorithm to guide the selection based on objective evaluation, refinement verification, and generation of initial selections.
- We present a black-box optimization flow that combines Bayesian optimization with local optimization to collaborate with contract-based system reasoning for optimizing the selection.
- Experimental results based on a UAV propulsion system design problem show that the proposed methodology outperforms the existing methods while satisfying all design goals.

The remainder of this chapter is organized as follows: Section 7.2 introduces black-box optimization. Section 7.3 defines the problem and gives an overview of our contract-based component selection algorithm. Section 7.4 details the contract-based system reasoning. Section 7.5 presents the black-box optimizer. Section 7.6 reports and analyzes the experimental results. Finally, Section 7.7 concludes the chapter.

7.2 Black-box Optimization

This section introduces black-box optimization.

Black-box optimization, also known as derivative-free optimization, is an optimization problem characterized by evaluations of function values and constraints without access to gradients or the use of gradient approximation [41]. In this paradigm, the algorithm for black-box optimization is constrained to depend on evaluations from previously explored points.

Techniques for black-box optimization can be categorized into two types: model-based methods and direct search methods. Model-based methods create a surrogate model to approximate the objective function and then perform optimization while refining the surrogate function. Bayesian optimization is a typical example where the function form is not assumed, and the surrogate model is constructed by the observed points. Direct search methods compare the evaluations of the previously explored points to determine the next exploration point. Examples of the methods include the Nelder-Mead algorithm, simulated annealing, and coordinate descent.

7.3 Contract-based Component Selection

In this section, we first give the problem formulation of the contract-based component selection problem and then introduce our proposed algorithm that combines a black-box optimizer with contract-based system reasoning.

7.3.1 Problem Formulation

The contract-based component selection problem is defined as follows:

Problem 7.1. *Given a system netlist describing the connections between the subsystems, a design specification, a selection objective, candidate components for each subsystem, and specifications of the candidate components, select a component for each subsystem such that the composed system satisfies the design specification and optimizes the selection objective.*

Here we detail the elements and their notations in the selection problem:

- *System Netlist*: A system netlist is a tuple $\mathcal{N} = (S, \mathcal{P}_s, T, \sigma, \Sigma, \rho, P_{sub}, E)$, where S is the set of all subsystems, $\mathcal{P}_s$ denotes the set of ports in the

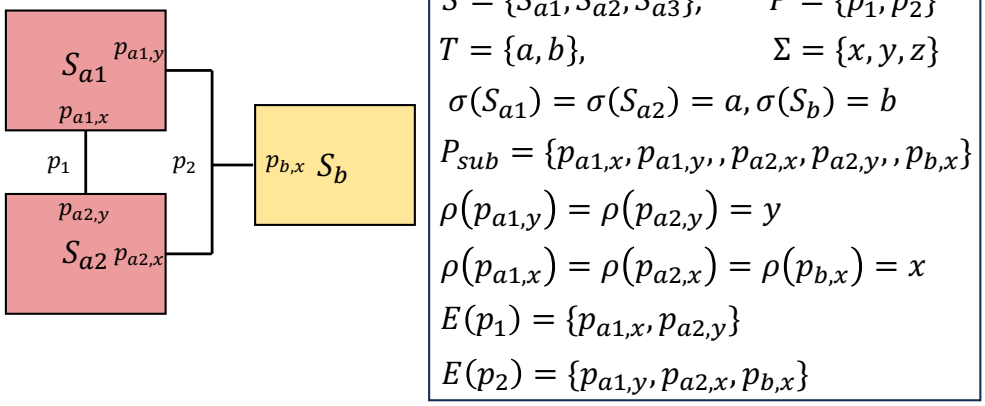


Figure 7.2: An example system netlist and the notations.

system, T is the collection of subsystem types in the system, $\sigma : S \mapsto T$ maps each subsystem to a subsystem type. Σ is the collection of variables in contracts to describe the behaviors, P_{sub} is the collection of ports in the subsystem. $\rho : P_{sub} \mapsto \Sigma$ maps a subsystem port to a variable in contracts. $E : \mathcal{P}_s \mapsto 2^{P_{sub}}$ describes the connection between subsystem ports and the corresponding system port.

- *Design Specification*: A design specification is an assume-guarantee contract $\mathcal{C}_S^{ag} = (A_S, G_S)$.
- *Candidate Components*: The candidate components for a subsystem type $t \in T$ is a library of contracts, defined as follows: $\mathcal{L}_t = \{\mathcal{C}_{t,j} = (A_{t,j}, G_{t,j}) = (C_{t,j}, B_{t,j}) \mid j = 0, ..., n_t - 1\}$, where n_t is the number of candidate components for the subsystem type. The assumption, guarantee, constraint, and intrinsic behavior of each candidate component are expressed using the variables in Σ. The set of all candidate components is the component library $\mathcal{L} = \bigcup_{t \in T} \mathcal{L}_t$.
- *Selection Objective* A selection objective is a pair (f, A_f), where $f : \mathcal{B}_{\mathcal{P}_s} \mapsto \mathbb{R}$ is the objective function, and A_f defines the environment to evaluate the objective function.
- *Selection of Components* A selection $\alpha : S \mapsto \mathcal{L}$ maps each subsystem to a candidate component.

Figure 7.2 shows an example of a system netlist consisting of three subsystems and two connections.

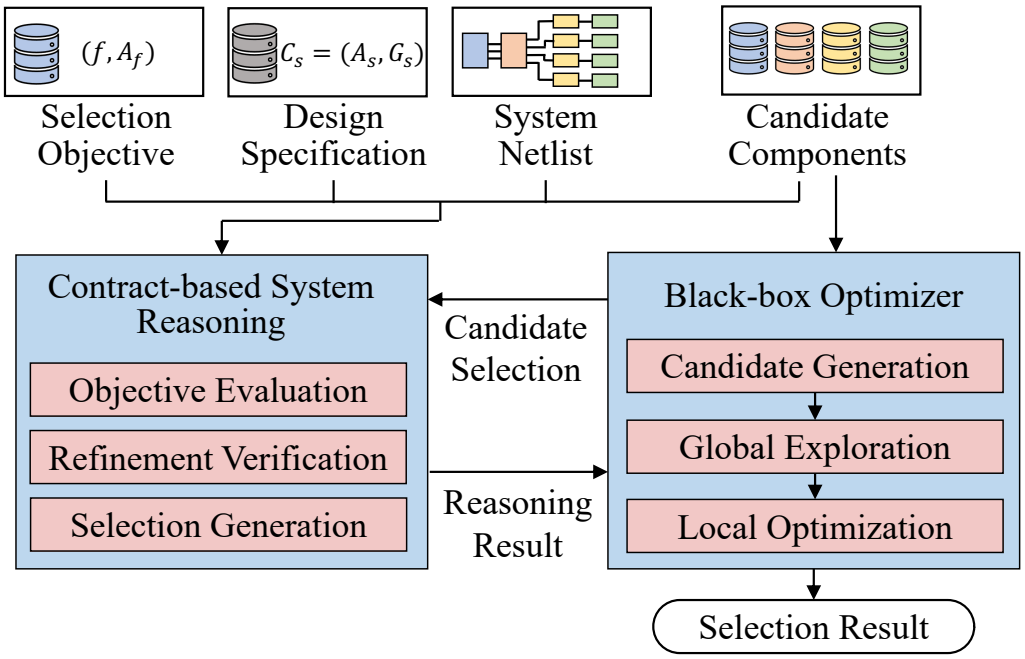


Figure 7.3: An overview of the proposed contract-based component selection flow.

7.3.2 Algorithm Overview

The contract-based component selection problem can be cast into the following optimization problem:

$$\min_{\alpha} \quad f(h) \tag{7.1a}$$

$$s.t. \quad h \in A_f \cap C_{comp} \cap B_{comp} \tag{7.1b}$$

$$(C_{comp}, B_{comp}) = (\|_{s \in S}\ \alpha(s)) \tag{7.1c}$$

$$(C_{comp}, B_{comp}) \preceq \mathcal{C}_S, \tag{7.1d}$$

where $(\|_{s \in S}\ \alpha(s))$ is the composition of all contracts based on the selection α, (C_{comp}, B_{comp}) is the composed contracts, and h is a behavior.

To solve the optimization problem, we propose a contract-based component selection flow consisting of two parts: *Contract-based System Reasoning* and *Black-box Optimizer*. Figure 7.3 shows the overview of the flow.

The contract-based system reasoning performs contract operations on the design specification and the candidate components to guide the black-box optimizer with three tasks: (1) *Objective Evaluation* finds the behavior and evaluates the objective of a candidate selection. (2) *Refinement Verification* verifies whether a candidate selection meets the design specification using contract refinement. (3) *Selection Generation* generates selections of components for the black-box optimizer to explore the selection space.

The black-box optimizer optimizes the selection by interacting with the contract-based system reasoning in three stages: (1) *Candidate Generation* first requests the selection generation task to generate initial selections. (2) *Global Exploration* then utilizes Bayesian optimization to explore the selection space and collect top-performing designs. (3) *Local Optimization* refines the top-performing designs and returns the selection result by optimizing the selection for one subsystem iteratively.

7.4 Contract-based System Reasoning

Contract-based design eases the designer's burdens by leveraging contract operations on the specifications of individual components. Managing contracts and performing contract operations for Equation 7.1 are crucial in contract-based component selection. This section describes contract-based system reasoning that supports contract operations and system reasoning to assist the black-box optimizer.

As shown in Figure 7.4, contract-based system reasoning consists of three steps: (1) *Contract System Creation* generates constraints based on the relation between ports and subsystem types in the system netlist. (2) *SMT Clause Encoding* converts the generated constraints and contracts of the selected components into SMT formulas. (3) *Task Execution* performs the tasks to evaluate the objective, verify refinement, and generate selections in response to the requests from the black-box optimizer.

Without loss of generality, in this chapter, we assume the candidate components of the library are specified in constraint-behavior contracts and the design specification is an assume-guarantee contract.

7.4.1 Contract System Creation

The input to the selection problem defines the constraints for subsystem composition and selection space. The system netlist describes composition rules based on the connections, and the candidate components define the selection space. This step generates system constraints S_C and selection constraints S_{sel}. The system constraints represent composition rules to ensure that connected system ports and subsystem ports have the same behaviors. The selection constraints encode the selection space by introducing auxiliary variables to denote a selection of a candidate component for each subsystem. These constraints are reused in task execution as the system netlist remains constant.

Algorithm 8 summarizes the contract system creation. Lines 4–13 generate the selection constraints by enumerating available candidate components for each subsystem. Line 8 creates a Boolean auxiliary variable $u_{s,i}$ whose value indicates whether a candidate component $\mathcal{C}_{t,i}$ is selected for a subsystem s. Lines 10 and 11 ensure the activation of the selected components in the system. Line 12 stores the auxiliary variables for subsequent steps. Line 13 ensures the validity of the selection by restricting that each subsystem selects exactly one candidate component. For example, we create the following constraints if the subsystem has

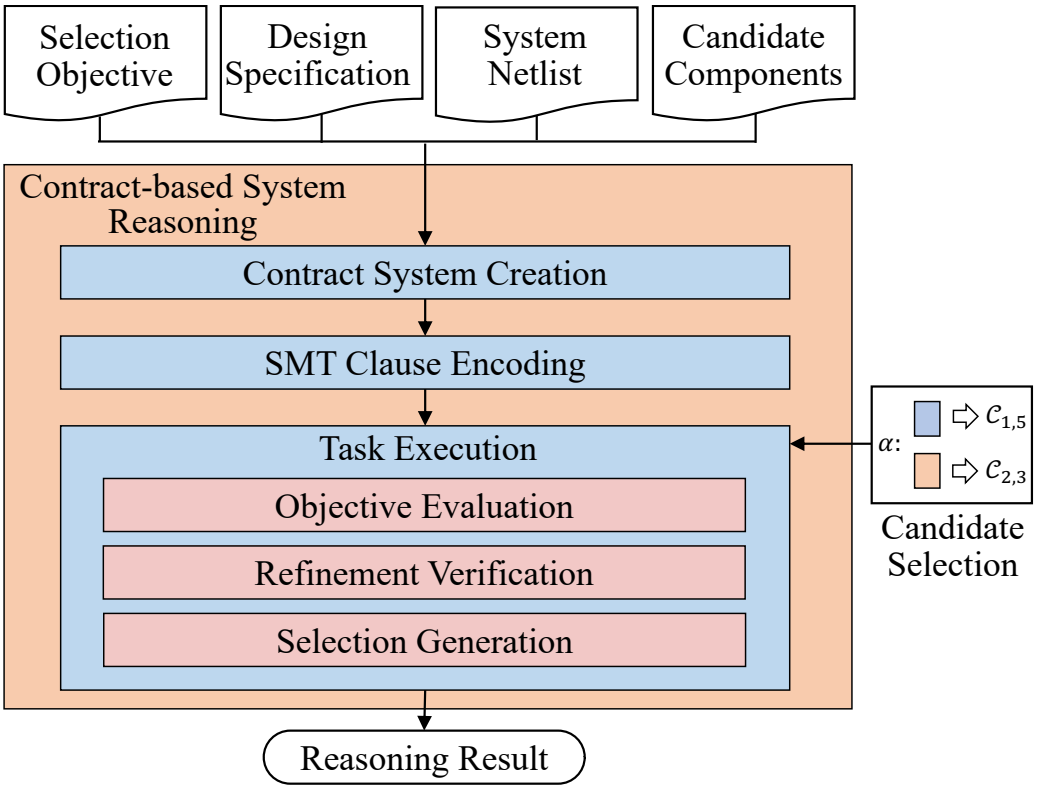


Figure 7.4: An overview of contract-based system reasoning.

three candidate components $U_s = \{u_{t,1}, u_{t,2}, u_{t,3}\}$:

$$u_{t,1} + u_{t,2} + u_{t,3} = 1.$$

Lines 14–16 generate the system constraint by enforcing the equivalence of the connected port behaviors.

7.4.2 SMT Clause Encoding

This step encodes the contracts and the generated constraints as SMT formulas to leverage SMT solvers for system reasoning. We ensure no duplicate variables for the same candidate components in different subsystems in this step. First, we create a rename function that maps the symbols Σ to the subsystem ports P_{sub} based on the mapping ρ. We then copy the contracts of the candidate components, create a new variable for each subsystem port, and then replace the variables Σ in the candidate components with the new variables. Finally, the constraints and contracts are encoded into SMT formulas based on the background theory for describing the behavior.

Algorithm 8 Contract System Creation

Inputs: System netlist $\mathcal{N} = (S, \mathcal{P}_s, T, \sigma, \Sigma, \rho, P_{sub}, E)$, design specification $\mathcal{C}_S^{ag}$, and candidate components $\mathcal{L}$.
Output: System constraints S_C, selection constraints S_{sel}, and selection variable maps U.

```
1: S_C ← ∅
2: S_sel ← ∅
3: U ← map()
4: for all subsystem s ∈ S do
5:     t = σ(s)
6:     U_s = ∅
7:     for i = 0, i < n_t − 1, i = i + 1 do
8:         u_{t,i} = new_variable()
9:         U_s ← U_s ∪ {u_{t,i}}
10:        S_sel ← S_sel ∪ imply(u_{t,i}, C_{t,i})
11:        S_sel ← S_sel ∪ imply(u_{t,i}, B_{t,i})
12:    U[s] = U_s
13:    S_sel ← S_sel ∪ exact_select_one(U_s)
14: for all system port p ∈ P_s do
15:    for all subsystem port p_s ∈ E(p) do
16:        S_C ← S_C ∪ equal_value(p_s, p)
17: return S_C, S_sel
```

In the following sections, the notations for constraints, contracts, and selection represent the encoded SMT formulas.

7.4.3 Task Execution

This step performs the tasks requested by the black-box optimizer.

7.4.3.1 Objective Evaluation

This task evaluates the objective based on the valid behavior of the composition of the selected components. We say a behavior is valid if the behavior satisfies the constraints C of the constraint-behavior contracts. Therefore, a valid behavior can be found by the SMT formula $A_f \wedge C_{comp} \wedge B_{comp}$, where (C_{comp}, B_{comp}) is the constraint-behavior contract of the composed system.

Algorithm 9 details the task. Lines 1–4 compose the contract by creating the conjunction of the SMT formulas from the environment of the selection objective, the system constraints, and the contracts of the selected components. Line 5 invokes the SMT solver to solve the generated SMT formula. The solver returns the satisfiability r_{sat} and a valid behavior h based on the satisfiable assignment. Finally, Line 6 returns the objective function value. We assume that A_f guarantees the same $f(h)$ value across all valid behaviors satisfying the formula l_c, ensuring a meaningful evaluation even if the component contract is refined. Different $f(h)$ values under the selection objective require evaluating all possible $f(h)$ values.

Algorithm 9 Objective Evaluation

Inputs: System constraints S_C, design specification $\mathcal{C}_S^{ag} = (A_S, G_S)$, candidate components $\mathcal{L}$, selection objective (f, A_f), and selection of components α.
Output: The objective function value $f(h)$ based on a valid behavior h of the selection.

1: $l_c \leftarrow A_f \wedge S_C$
2: **for all** subsystem $s \in S$ **do**
3: $\quad (C_s, B_s) = \alpha(s)$
4: $\quad l_c \leftarrow l_c \wedge B_s \wedge C_s$
5: $r_{sat}, h \leftarrow$ `SMT_solve`(l_c)
6: **return** $f(h)$

Algorithm 10 Refinement Verification

Inputs: System constraints S_C, design specification $\mathcal{C}_S^{ag} = (A_S, G_S)$, candidate components $\mathcal{L}$, and selection of components α.
Output: A truth value indicating whether the composition of the selected components refines the design specification.

1: $l_c \leftarrow A_S \wedge S_C$
2: $r_c \leftarrow G_S$
3: **for all** subsystem $s \in S$ **do**
4: $\quad (C_s, B_s) = \alpha(s)$
5: $\quad l_c \leftarrow l_c \wedge B_s$
6: $\quad r_c \leftarrow r_c \wedge C_s$
7: $r_{sat}, h \leftarrow$ `SMT_solve`$(l_c \wedge \neg r_c)$
8: **return** not r_{sat}

However, contract refinement, which restricts the behaviors, may result in a subset of these possible values, undermining the evaluation's effectiveness.

7.4.3.2 Refinement Verification

This task verifies the design specification and compatibility of the subsystems by checking the refinement relation with the design specification, which can be converted to an SMT formula $(A_S \wedge B_{comp}) \wedge (\neg G_S \vee \neg C_{comp})$. Intuitively, $A_S \wedge B_{comp}$ represents the possible behaviors when the composed system functions normally in any environment satisfying A_S. This set of behaviors should satisfy the guarantee G_S to meet the design specification. In the case of incompatible subsystems, the violation of constraints C_{comp} implies that the intrinsic behaviors no longer hold. Therefore, the SMT formula is satisfiable if any behavior in $A_S \wedge B_{comp}$ violates the constraints C_{comp} or the guarantee G_S. This behavior serves as a counter-example, indicating that the selected components do not meet the design specifications since the refinement relation does not hold.

Algorithm 10 outlines the task. In Lines 1–6, two conjunctions of SMT formulas, l_c and r_c, are generated from the inputs. Notably, l_c corresponds to $(A_S \wedge B_{comp})$ while r_c is the negation of $(\neg G_S \vee \neg C_{comp})$. Lines 7–8 invoke an SMT solver and return the verification result.

Algorithm 11 Selection Generation

Inputs: System constraints S_C, selection constraints S_{sel}, design specification $\mathcal{C}_S^{ag} = (A_S, G_S)$, candidate components $\mathcal{L}$, selection objective (f, A_f), lower bound performance L_b.
Output: The selection of components α.

```
1: l_c ← A_f ∧ S_C ∧ SMT_clause(f > L_b)
2: r_sat, h ← SMT_solve(l_c ∧ S_sel)
3: α = find_selection(U, h)
4: while resource_left() and r_sat do
5:    if refinement_verification(S_C, , C_S^ag, L, α) then
6:       L_b ← value(f)
7:       l_c ← l_c ∧ SMT_clause(f > L_b)
8:    else
9:       S_sel ← S_sel ∧ not_select(α)
10:   r_sat, h ← SMT_solve(l_c ∧ S_sel)
11:   α = find_selection(U, h)
12: return α
```

7.4.3.3 Selection Generation

While the preceding two tasks offer information for a given selection, they do not actively guide optimization by suggesting selections. A selection of components that meets the design specification can serve as an initial solution and provide a lower bound for selection optimization. This task, therefore, aims to generate selections that satisfy the design specification. In contrast to the approach in [133], we extend the specifications to contracts, allow optimization for an objective, and ensure satisfaction of the design specification.

Algorithm 11 summarizes the selection generation. Line 1 composes the contracts to generate an SMT formula from the design specification, the selection objective, and a lower bound L_b on the objective function value. Lines 2–11 generate selections until no better selections can be found or the resources allocated for the tasks are used up. The satisfiability of the SMT formula $l_c \wedge S_{sel}$ indicates whether a selection can be found to produce a behavior whose objective value exceeds the lower bound. Line 5 checks whether the selection meets the design specification. If the design specification is verified, Lines 6–7 update the lower bound value to optimize the selection. Otherwise, Line 9 removes the selection from the selection space.

As an SMT problem might be undecidable, its execution time is unbounded, and the termination of the solver is not assured. To address this, the added resource allocation mechanism terminates the task if the problem becomes too challenging for the solver. As will be introduced in the next section, the generated selections are used to produce initial solutions within the specified time limits.

The selection generation returns the optimal solution if the selection space is finite and resources are unlimited. This is because the optimal selection must satisfy the SMT formula, meet the design specifications, and have the largest value L_b among all selections that satisfy the design specification. When the optimal selection has not been reached, the SMT must remain satisfiable since its objective function value exceeds all current lower bounds. As the

lower bound strictly increases with each iteration, the finite selection space guarantees that the loop will eventually terminate at the optimal objective function value. Consequently, the optimal solution is ensured.

7.4.4 Complexity Analysis

This section analyzes the complexity of our contract-based system reasoning. We define n_c^{max} as the maximum number of formulas among the constraints, n_b^{max} as the maximum number of formulas among intrinsic behavior, n_{A_f} as the number of clauses in A_f, n_{A_S} as the number of clauses in A_S, n_{G_S} as the number of clauses in G_S, n_t^{max} as the maximum number of candidate components for a subsystem type, and n_{iter} as the number of iterations in the loop in Algorithm 11.

7.4.4.1 Number of Variables

The variables within the SMT formulas originate from two sources: those representing the values of ports and the auxiliary selection variables. The number of auxiliary selection variables is $O(|S||n_t^{max}|)$, and the number of variables for port values is $|P|$. Consequently, The SMT formulas for objective evaluation and refinement verification contain $|P|$ variables, while selection generation requires $O(|P| + |S||n_t^{max}|)$ variables.

7.4.4.2 Number of Clauses

First, we analyze the number of clauses for system constraints S_C and selection constraints S_{sel}. The number of clauses in S_C is $O(|P|^2)$ because of the equivalent constraints created in Line 16 of Algorithm 8. Similarly, the number of clauses in S_{sel} is $O(|S||n_t^{max}|)$ as each candidate component for a subsystem requires a clause to indicate the selection.

Next, we analyze the number of clauses in the three proposed tasks. The number of clauses in objective evaluation is bounded by $O(n_{A_f} + |P|^2 + |S|(n_c^{max} + n_b^{max}))$, which accounts for the clauses in A_f, S_C, and the candidate component contracts. Similarly, the number of clauses in refinement verification is bounded by $n_{A_S}+|P|^2+n_{G_S}+|S|(n_c^{max}+n_b^{max})$, considering the clauses in A_S, G_S, and S_C, and the candidate component contracts. Finally, the number of clauses in each SMT formulate in the selection generation is bounded by $O(n_{A_f} + |P|^2 + |S|^{|n_t^{max}|} + n_{iter})$, as each iteration adds a clause to update the lower bound or remove the selection from selection space.

7.5 Black-box Optimizer

The tasks of the proposed contract-based reasoning system have covered all necessary operations for the optimization problem in Equation 7.1. Therefore, the optimization problem can be treated as a black-box optimization problem, aiming to find the optimal selection α,

where the objective function value and constraint satisfaction rely on contract-based system reasoning. In this section, we present our black-box optimizer.

7.5.1 Candidate Generation

Satisfying optimization constraints is challenging for a black-box optimizer, as the optimizer has no prior knowledge of how to satisfy the constraints. Therefore, initial candidate selections are important as they can guide the optimizer by the objective values of the valid selections and the selection that does not meet the design specification. In this step, the black-box optimizer invokes the selection generation task in contract-based system reasoning to obtain initial candidate selections. To prevent potential long execution times resulting from the undecidability and high complexity of the SMT problem, a time limit t_{max} and a maximum number of iterations n_{cg} are set to ensure the predictable termination of Algorithm 11. All generated selections, including the valid selections and those that fail the refinement verification, are retained to guide the selection process.

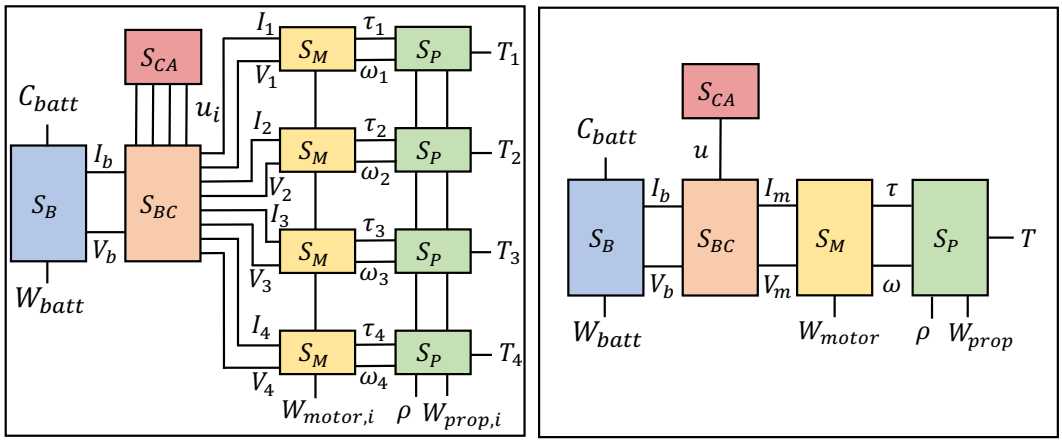
7.5.2 Global Exploration

Global exploration is a crucial step to find the optimal selection and ensure the optimization is not limited to a local minimum. Therefore, Bayesian optimization is a promising candidate as an exploration algorithm since it automatically performs tradeoffs between exploration of the selection space and exploitation of good selections. We optimize the selection of components in Bayesian optimization by using categorical variables to represent the selection of components. Each categorical variable corresponds to a selection for a subsystem, and the candidate components are the allowable values for the variable. The tree-structured Parzen estimator (TPE) [23] is chosen as the surrogate model for its capability to handle categorical spaces.

The process begins by creating an initial surrogate model using the candidate selections of the previous step. Subsequently, Bayesian optimization uses the surrogate model to explore the selection space, stopping after a maximum of n_{bo} iterations. Finally, the top-performing k candidate selections are retained for local optimization. The parameters k and n_{bo} are designer-defined, allowing for tradeoffs between exploration time and exploration range.

7.5.3 Local Optimization

This step iteratively refines the top-performing candidates through local optimization and returns the one with the largest objective function value. In each iteration, we randomly choose a subsystem and then optimize the selection of the subsystem by enumerating the available components. This iterative process continues until no further improvement is achieved or the number of iterations exceeds n_{lo}, a designer-defined parameter.



(a) UAV Propulsion System (b) System with Symmetry

Figure 7.5: An example system netlist and the simplified system netlist of the UAV propulsion system with one battery, four motors, and four propellers, where S_B denotes batteries, S_{CA} is a control algorithm, S_{BC} is a battery controller, S_M represents motors, and S_P denotes propellers.

7.6 Experimental Results

To show the effectiveness and efficiency of our contract-based component selection algorithm, we implemented our approach in Python with Z3 [48] as the SMT solver and Optuna [5] for Bayesian optimization with the TPE surrogate model. The background theory used in the SMT formula is polynomial arithmetic. For the TPE surrogate model, hyperparameter settings include 50 warm-up iterations, with other iteration-dependent parameters following the default generation function in Optuna [5]. The evaluation platform was an Intel I9-9980HK machine with 32GB memory.

Table 7.1: Statistics of the test cases, including number of motors (N_m), number of batteries (N_b), and weight of the UAV frame (W_{body}) and comparisons of the objective function values (OV), satisfactions of the design specification (DS), and runtimes (sec) for our proposed method with the baseline method.

Design Netlist				Our method			[73]			[133]			[129]		
	N_m	N_b	W_{body}	OV	DS	Runtime	OV	DS	Runtime	OV	DS	Runtime	OV	DS	Runtime
netlist1	4	2	2	1.225	O	1589.97	NA	X	> 18000*	0.721	O	> 18000	0.673	O	> 18000*
netlist2	4	3	2	1.220	O	1239.61	1.220	O	7619.82*	0.154	O	> 18000	0.757	O	> 18000*
netlist3	6	3	2	1.393	O	1834.28	1.345	O	6302.93*	1.128	O	> 18000	0.781	O	> 18000*
netlist4	4	1	2	1.076	O	1553.68	NA	X	> 18000*	0.063	O	> 18000	0.646	O	> 18000*
netlist5	4	2	5	0.666	O	1571.04	NA	X	> 18000*	NA	X	> 18000	0.367	O	> 18000*

* The runtime does not include the time for manual reformulation of the objective function.

Table 7.2: An example that shows the satisfaction of the system specification is not monotonic to the parameters. The selected battery is TurnigyGraphene6000mAh6S75C, and the system netlist is netlist1.

Propeller \ Motor	KDE3520XF_400	P60KV170
apc_12x6SF	O	X
apc_18x12E	X	O

The test cases were extracted from a UAV propulsion system design challenge [171, 44]. Table 7.1 summarizes the test cases statistics, while Figure 7.5 illustrates an example system netlist of the design. The system comprises batteries, motors, propellers, a battery controller, and a control algorithm. We simplified the design by leveraging symmetry to combine batteries, motors, and propellers, as shown in Figure 7.5(b). The number of candidate components is 348 for propellers, 146 for motors, and 56 for batteries, leading to a selection space of nearly three million combinations. The contract for each candidate component can be found in Chapter 4.

The design specification $\mathcal{C}^{ag}_{fly} = (A_{fly}, G_{fly})$ requires that the UAV operate normally under the maximum voltage of the selected battery:

$$\begin{aligned} A_{fly}:\ & \rho = 1.225, u = 1 \\ & W_s = W_{batt} + W_{body} + W_{prop} + W_{motor} \\ G_{fly}:\ & T_s \geq W_s, \end{aligned}$$

where W_{body}, an input from the design, is the weight of the UAV frame.

The selection objective of the design (f_{hover}, A_{hover}) is to optimize the hovering time:

$$\begin{aligned} f_{hover}:\ & \frac{C_{batt}}{I_b} \\ A_{hover}:\ & T_s = W_s = W_{batt} + W_{body} + W_{prop} + W_{motor}. \end{aligned}$$

We set the designer-defined parameters t_{max} to 100 seconds, n_{cg} to 200, n_{ge} to 1000, k to 10 and n_{lo} to 5.

We compared our proposed algorithm with those of Iannopollo *et al.* [73], Peter *et al.* [133] and Oh *et al.* [129]. The approach in [113], however, requires all components to be expressed in Hoare logic, making it inapplicable to general CPS designs, such as our test cases, which include physical components using constraint-behavior contracts. Although the approach was not included in our experiments, we find it reasonable to expect that the comparison results with [133] would similarly apply to their work, given the absence of optimization considerations.

Due to different assumptions or reliance on parameters-based optimization in these methods, we adapted the selection problem and their algorithms to ensure their applicability to our test cases for fair comparisons. First, the algorithm proposed in [73] explicitly assumes the objective is independent of the formalism used to describe the specification of components, which does not align with our test cases. For example, f_{hover} depends on I_b, the port value determined by the component contracts and the system behaviors. To adhere to this assumption, we manually reformulated f_{hover} as a function of the parameters by combining the contracts and the system netlist. The following equation shows the reformulated objective for netlist1:

$$f_{hover} = \frac{2C_{batt}}{4(\frac{C_p D r}{4C_t 2\pi K_t} + I_{idle} r)(\frac{C_p D r}{4C_t 2\pi K_t} + I_{idle} r + \frac{2\pi}{K_v D^2}\sqrt{\frac{W_s}{4\rho C_t}})}. \tag{7.2}$$

Given that the objective function is neither convex nor linear in its parameters and the parameters are defined on a categorical space defined by the components, Bayesian optimization was chosen as the discrete optimizer.

The selection algorithm proposed in [133] focuses on finding a feasible solution for the system specification rather than performing optimization. We have extended their method as an optimization flow in our tasks *Selection Generation* in Section 7.4.3.3. Therefore, we used our selection generation to represent their method in the experiment.

The work [129] focused on parameter synthesis for parametric stochastic contracts. Consequently, we must reformulate the selection objectives for their algorithm since it only applies to objective functions expressed by parameters. Moreover, their proposed method relies on a monotonic property of the refinement relation to reduce the exploration space. The parameters in parametric stochastic contracts monotonically affect its refinement relation to the system, as the satisfaction of the chance constraint is monotonic if we relax a constraint. This property motivates them to partition the design space, effectively reducing the exploration space. However, it's important to note that the general component selection problem doesn't inherently possess this property. As evidenced in Table 7.2, a change in motor selection from P60KV170 to KDE3520XF_400 alters the satisfaction of the system specification from unsatisfaction to satisfaction for propeller apc_12x6SF, while the same change in motor transitions the satisfaction of the system specification from satisfaction to unsatisfaction for propeller apc_18x12E. The absence of the properties results in the inability to guarantee satisfaction in the partitioned selection space. Consequently, the method becomes an exhaustive search as the partitioning must yield the set of all selection combinations.

Table 7.1 lists the comparison results and the statistics about the test cases. Overall, our proposed method successfully selected the components to meet the design specification in all test cases, while [73] and [133] failed to complete some test cases within five hours. Compared to the methods in [129], our method achieves an average 73.9% longer hovering time for the test cases that successfully generate a valid selection.

The reasons why our method outperforms the previous works are as follows:

- In [73], the method optimizes the objective before verifying the system specification based on the optimization result, leading to explorations of selections with high objective values that fail to meet the system specification. As a result, the prolonged execution time hinders its effectiveness in finding a feasible selection within the time limit, despite its potential to produce comparable selection results once a feasible selection is found. In contrast, our approach integrates refinement verification into the optimization process using contract system reasoning, thus achieving high selection quality and reducing execution time.

- When the SMT solver encounters a challenging formula, the optimization procedure may fail to produce a valid selection within a reasonable time. The method in [133] has to wait for the termination of the SMT solver, resulting in low selection quality due to insufficient optimization iterations within the time limits. On the contrary, our proposed method avoids long execution time by introducing a resource allocation mechanism.

- The method in [129] degenerates to an exhaustive search due to the lack of monotonicity in satisfying the system specification. In contrast, our algorithm combines global exploration with local optimization to adequately explore the selection space and reach a local optimum within reasonable runtimes.

- Last but most importantly, reformulating the selection objective based on the netlist and contracts is necessary to apply the approach in [73] and [129]. This task requires designers to maintain complex mathematical formulations such as Equation (7.2), as discussed in Section 7.1. In contrast, our proposed flow utilizes behaviors for selection, fully leveraging contract-based design methodology. Consequently, our flow automates the component selection process, eliminates the need for manual reformulation, and achieves high-quality results and efficient exploration without relying on oversimplified assumptions.

7.7 Conclusion

We presented a contract-based component selection flow using behaviors. The proposed contract-based system reasoning effectively assists the black-box optimizer in exploring the selection space with contract operations. Experimental results demonstrated that the proposed flow outperforms the existing methods. Our potential future works include: (1) developing a codesign paradigm to optimize the system netlist and component selection using behaviors, as system design problems often require finding the connectivity between subsystems; and (2) enhancing exploration efficiency by leveraging the contract formalisms and pruning selection space based on the compatibilities between subsystems.

Chapter 8

ContractDA: An Automation Tool for Contract-based Design

As introduced in Chapter 3, automation tools for contract-based design are crucial for enabling the adoption of contract-based design methodology by designers who are not experts in contract operations. However, existing tools provide only a subset of automation tasks, making it difficult to adapt to diverse applications and hindering further research. This chapter introduces ContractDA, a new tool designed to provide comprehensive automation support for contract-based design. The tool covers key tasks, including specification, verification, simulation, and synthesis, while also offering various contract manipulations to efficiently manage and operate on contracts. Additionally, ContractDA features both command-line inputs and a Python API, ensuring flexibility in its usage and extensibility to accommodate new formalisms and algorithms.

8.1 Introduction

Contract-based design, originated from software engineering for specifying programs [109], has emerged as a promising design methodology for addressing CPS design challenges through contracts, a type of formal specifications [17, 20, 151], and their systematic manipulation. Contract manipulations enable the decomposition of top-level system specifications into manageable subsystem specifications, and the separation of different design aspects. Such decomposition and separation reduce design complexity, leading to improved design efficiency. Moreover, the use of formal specifications and their rigorous relations, such as refinement and contract replaceability, support correct-by-construction design, ensuring that the final implementation satisfies the top-level specification even after multiple layers of decomposition.

To support the adoption of contract-based design in practice, theories and algorithms have been developed to enhance the quality and efficiency of the decomposition process. Contract theories focus on the fundamental properties of contracts, leading to closed-form

formulas for contract manipulations and a deeper understanding of their role in system reasoning [15, 17, 77]. Algorithms for contract-based design, including verification [35, 76, 96] and synthesis [73, 130, 153, 173], leverage these theoretical foundations to enable essential design steps.

Despite the advancements in theories and algorithms for contract-based design, automation support remains fragmented and limited, making it difficult to adapt to diverse applications and hindering further research. As introduced in Chapter 3, existing tools [34, 40, 73, 79, 106, 124, 153] provide only a subset of automation tasks, depending on their specific development context. The lack of comprehensive automation support requires designers to familiarize themselves with multiple tools and switch between them to complete design tasks. Moreover, limited support for contract manipulations, including contract operations, relations, and properties, restricts the full potential of contract-based design. For instance, designers cannot easily identify missing viewpoints in a design without an operation like separation [132]. Additionally, automation tools should offer a balance between ease of use and versatility, providing simple commands for straightforward tasks while supporting programmability for handling complex design challenges. Given these limitations in existing tools, both practical applications and research on contract-based design often rely on custom-built tools [121, 148, 164], which creates barriers to wider adoption and further methodological advancements.

As a result, developing automation tools that address these concerns is essential to advance contract-based design research and its practical applications. This chapter introduces ContractDA, a new tool designed to provide automated support for contract-based design tasks and contract manipulations. The contributions of ContractDA are summarized as follows:

- The tool provides comprehensive support for contract-based design automation tasks, including specification, verification, simulation, and synthesis, allowing designers to rely on a standalone tool instead of navigating multiple independent tools. To the best of our knowledge, this is the *first* contract-based design tool that integrates functionalities for all these tasks.

- The tool also supports a comprehensive set of contract manipulations, enabling designers and researchers to efficiently manage and operate on contracts. A contract can be expressed with a single scenario or a viewpoint on one component, and then systematically combined through manipulations, allowing for easier management and examination of contract relationships.

- The tool provides both a command line interface (CLI) and an application programming interface (API), offering flexibility for different use cases. The CLI enables designers to quickly invoke the tool for specific design tasks without additional setup, while the API allows for complex and large-scale operations to be performed programmatically.

- The tool is publicly available open-source [180], with abstractions for the elements in contract theories. This approach encourages future research and allows tool extension to accommodate various contract formalisms, expressions, and algorithms.

The remainder of the chapter is organized as follows: Section 8.2 introduces the functionality provided by the tool. Section 8.3 details the design of the tool. Section 8.4 provides an overview of the tool's usage in the contract-based design framework. Section 8.5 presents the practical experience with the tool in design problems and contract research. Finally, Section 8.6 concludes the chapter.

8.2 Functionality

The main functionality of ContractDA can be separated into system design-level tasks and contract manipulations. System design-level tasks refer to the automation tasks introduced in Chapter 3. These tasks perform specific steps in contract-based design, such as decomposing a specification into multiple subsystems via synthesis, checking the correctness of the decomposition through verification, or examining whether contracts match the design intent through simulation. Contract manipulations focus on contract theories to provide operations, check relations, and obtain the properties of contracts. With these fundamental tasks, designers can write contracts in a straightforward manner, such as specifying a condition for a single component within a particular design viewpoint, and then incorporate these contracts into the overall system specification. Notably, these tasks also serve as the building blocks for the system design-level tasks.

ContractDA can be used via an interactive shell, input scripts, or Python APIs, offering flexibility for both straightforward tasks and programmability. The interactive shell and input scripts are designed for system-level tasks, enabling designers to invoke the tool for specific design steps. Python APIs support both system-level tasks and contract manipulation, allowing designers to address complex design challenges through programming. The tool supports two contract formalisms: assume-guarantee contracts and constraint-behavior contracts The tool currently supports the language of nonlinear arithmetic over real numbers, enabled by Z3, for defining the sets used in contracts. Its functionality can be extended by introducing new set expressions within our design framework, as detailed in Section 8.3. The following section details the supported tasks.

8.2.1 System Level Design Tasks

System-level design tasks support managing contracts and the contract-based design process to facilitate the design of a system. A design consists of systems and connections between system ports, defined either through a JSON format design file or dynamically via a Python API. Systems are associated with contracts that specify their behaviors. Each system can be described as several subsystems integrated through connections, with the contracts for the subsystems representing a decomposition of the system contract.

System-level design tasks operate on designs or systems by managing decomposition, handling connections, and performing contract manipulations to achieve the task. The supported tasks include:

- `verify_consistency` and `verify_compatible`, which check whether the environment and implementation sets for all contracts in a given design or system are non-empty. These tasks allow designers to validate contracts, preventing errors from meaningless contracts.
- `verify_refinement`, which verifies whether contract refinement relations hold for a given design or system. For a system, it checks whether the composition of all subsystem contracts refines the system contract. For a design, it recursively verifies all systems to ensure refinement relations hold throughout the design process.
- `verify_receptiveness`, which checks whether all contracts in a given design or system are receptive. This task is particularly relevant for domains requiring receptiveness in implementation, such as control systems and sequential programming.
- `verify_independent`, which verifies whether contract decompositions are correct for independent design. For a system, it checks whether its subsystem contracts form a valid decomposition. For a design, it recursively verifies all systems.
- `synthesis_component_selection`, which, given a system with defined subsystem connections but without contracts, a component library, and a synthesis objective, synthesizes the subsystems using the component library to satisfy the system contract.
- `simulate_automated`, which simulates a given system's contracts and produces critical behavior collections, as introduced in Chapter 6, enabling designers to review behaviors and verify design intent.
- `simulate_behavior`, which, given a system contract and defined environments, simulates the contract and returns behaviors that satisfy the environment contracts.

8.2.2 Contract Manipulations

Contract manipulation focuses solely on contracts, without considering connections and ports. The same variable name appearing in different contracts should denote the same variable and implicitly indicate a connection. The functionality in this category, summarized and compared with the existing tools in Table 8.1, includes all contract operations, relations, and properties, along with our developed contract replaceability introduced in Chapter 5. Consequently, this tool is the only one that provides a comprehensive set of contract manipulations.

These manipulation functionalities facilitate contract writing and support system-level tasks. For contract writing, since all operations are supported, designers can specify viewpoints and scenarios for a component and then combine them through operations, simplifying

Tools		OCRA	CONDEnSe	PyCo	AGREE	CHASE	CHROME	Pacti	ContractDA
Properties	Consistency	O	X	O	X	X	O	X	O
	Compatibility	O	X	O	X	X	O	X	O
	Receptivity	X*	X	X	X	X	X	X	O
Operations	Composition	X	O	O	X	O	O	O	O
	Quotient	X	X	X	X	X	X	O	O
	Conjunction	X	X	X	X	O	O	X	O
	Implication	X	X	X	X	X	X	X	O
	Merging	X	X	X	X	X	X	O	O
	Separation	X	X	X	X	X	X	X	O
	Disjunction	X	X	X	X	X	X	X	O
	Coimplication	X	X	X	X	X	X	X	O
Relations	Refinement	O	O	O	O	O	O	O	O
	Conformance	X	X	X	X	X	X	X	O
	Strong Dominance	X	X	X	X	X	X	X	O
	Strong Replaceability	X	X	X	X	X	X	X	O

* OCRA only checks if a given implementation is receptive to the contracts instead of checking receptiveness of contracts.

Table 8.1: Comparisons of the support for contract operations, properties, and relations of ContractDA and existing tools.

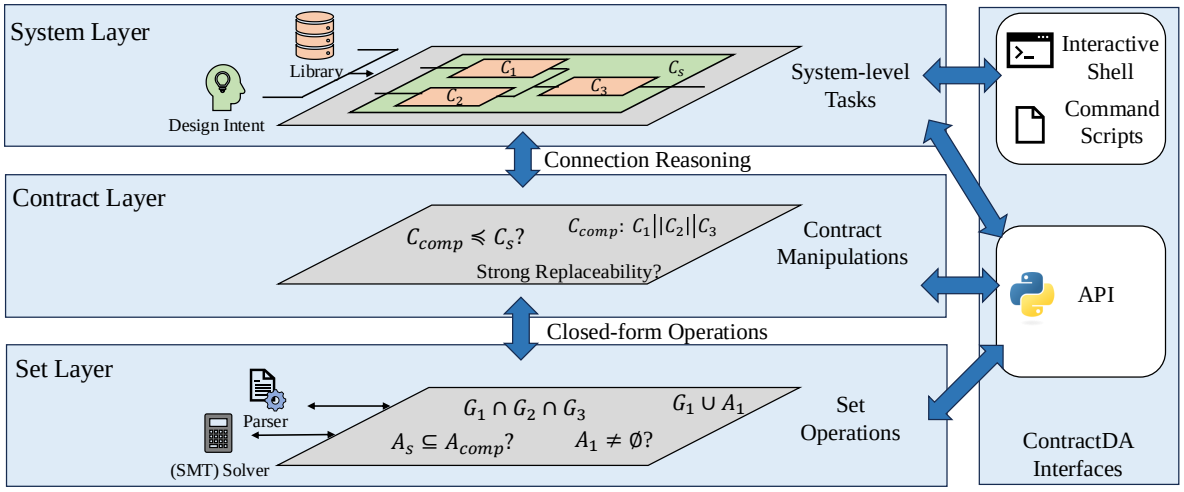


Figure 8.1: Overview of the design of ContractDA.

contract formulation without needing to handle everything at once. These functionalities are also used by system-level tasks to automate design steps.

With the provided functionality for system-level design and contract manipulations, users can perform various design tasks entirely within the tool.

8.3 Design of ContractDA

ContractDA is implemented in Python and currently supports assume-guarantee and constraint-behavior contracts expressed in first-order logic. To reason about first-order logic expressions, ContractDA interacts with Z3 [48] via its Python API. Reasoning is performed first by converting first-order logic expressions into SMT formulas, such as the set operations mentioned in Section 2.1.4.1 and algorithms introduced in the previous Chapters. After SMT solving, the satisfiability of the formulas and the satisfying assignments are converted back as the reasoning result.

Figure 8.1 illustrates the overview of the design of ContractDA. The tool consists of three main layers: set, contract, and system. The set layer forms the building blocks for contracts, as long as they support the required set operations. The set operations for contracts include intersection, union, subset relation, equivalence, and element query. These operations allow the tool to leverage set properties for contract reasoning. Furthermore, an abstraction class for set reasoning solvers is designed, enabling the integration of any solver that can take a set object and perform the required set operations.

The contract layer is built based on the set layer and adheres to contract theory. A contract is formed by combining the environment set and the implementation set. The supported assume-guarantee and constraint-behavior contracts can be defined through assumptions, guarantees, constraints, and intrinsic behaviors, while the environment and implementation are automatically inferred from these sets when needed. This ensures that the two contract formalisms can convert between each other and accommodate extensions for new contract formalisms. A contract is defined without any context of the systems, with variables representing the ports and connections implicitly defined by the same variables appearing in different contracts. This simplifies the process of manipulating contracts without the burden of resolving connections. The contract manipulation tasks are defined within this layer, enabling contract manipulation with an abstract design to accommodate future extensions.

The system layer describes specifications and manages the design process by incorporating decomposed subsystems, their connections, and the corresponding decomposed specifications. The system-level tasks within this layer automate the contract design process for verification, synthesis from libraries to optimize design objectives, and simulation to check the alignment of the design intent. The descriptions for systems, libraries, and design objectives can be read from a JSON file or constructed using the provided Python API. When executing system-level tasks, the contract description is converted into contracts in the contract layer. Connections between ports are then resolved by enforcing an additional constraint on both the environment and implementation sets after composition. For example, consider two assume-guarantee contracts: $\mathcal{C}_1 = (x \geq 1, y = 2x)$ and $\mathcal{C}_2 = (a \geq 2, b = 2a)$. The composition result, with a connection between port y and port a, is as follows: First,

the contract composition is performed on their saturation, resulting in:

$$\begin{aligned}\mathcal{C}_1 \parallel_{ag} \mathcal{C}_2 &= (A_{comp}, G_{comp}),\\ A_{comp} &= ((x \geq 1) \wedge (a \geq 2)) \vee \neg(y = 2x \vee x < 1) \vee \neg(b = 2a \vee a < 2),\\ G_{comp} &= (y = 2x \vee x < 1) \wedge (b = 2a \vee a < 2).\end{aligned}$$

Then, to ensure the connection always enforces the value of $a = y$, a constraint $C_{conn} = (a = y)$ is formed. The constraint is introduced to both the assumption and guarantee, resulting in:

$$\mathcal{C}_1 \parallel_{ag} \mathcal{C}_2 = (A_{comp} \cap C_{conn}, G_{comp} \cap C_{conn}).$$

This ensures the flexibility of a and y having different domains, while enforcing the SMT tool to consider only the conditions where y and a have the same value when reasoning over the sets.

The interface to ContractDA includes an interactive shell, scripts, and a Python API. The interactive shell is developed using the `prompt_toolkit` package [162]. Commands for the interactive shell are loaded during runtime. When the shell launches, it searches all source files in the directory where the command source file is located and registers the commands in the shell. This allows users to add custom commands to manipulate the tool, providing flexibility and extensibility. Scripts are executed as batch operations in the interactive shell. After running all the commands in a script, the interactive shell remains active, enabling interactions to obtain the resulting state and perform further processing. The Python API is defined following the three-layer separation, with each layer providing different functions to define systems. Abstraction for each layer is defined to accommodate extensions in future contract research.

With this design, the tool achieves the comprehensive functionality required for contract-based design while offering flexibility and extensibility for future research and improvements.

8.4 Contract-based Design with ContractDA

Figure 8.2 summarizes how ContractDA provides automation support for contract-based design. In the beginning, designers define the design intent, including specifications and objectives, and provide component libraries for use in the design. The goal is to generate an implementation that satisfies the specifications while optimizing the design objectives.

ContractDA provides the foundational support for entering the contract-based design framework by assisting designers in formulating contracts for specifications and objectives, This formulation is supported through assume-guarantee or constraint-behavior contracts, depending on the characteristics of the design target. The provided contract manipulations allow designers to specify contracts from individual viewpoints and conditions for each component, Once a contract is formulated, designers utilize simulation to verify its alignment

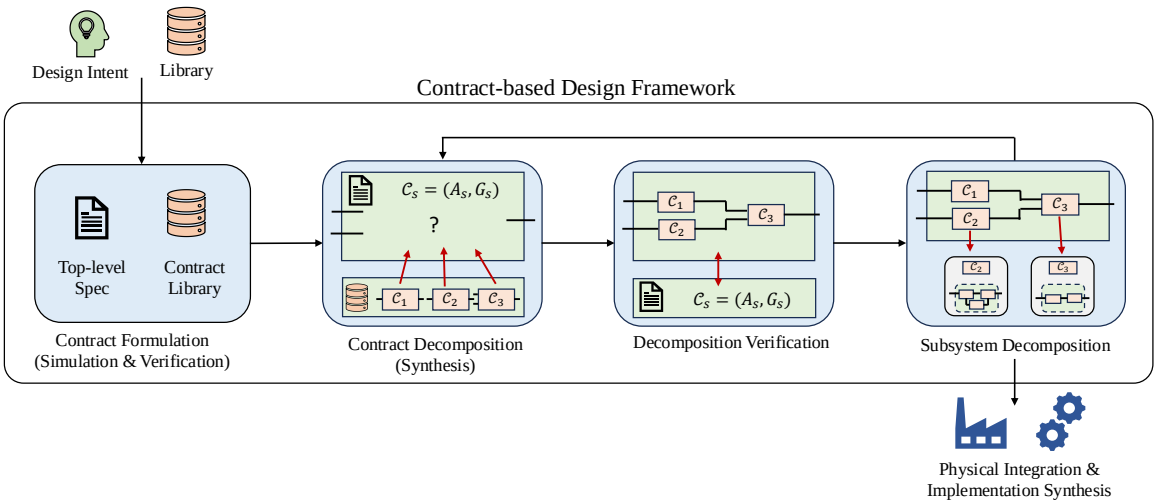


Figure 8.2: Usage of ContractDA in the contract-based design framework.

with the design intent. If the simulation results indicate a match, they can confidently proceed within the contract-based design framework. If a mismatch is discovered, designers revise the contract with the aid of problematic behaviors and expressions generated from the simulation results. Verification tasks, such as consistency and compatibility checks, ensure that the contract remains meaningful within contract semantics. The same process applies to the component library, ensuring that all elements entering the contract-based design framework correctly characterize both the design intent and component behaviors.

The contract-based design framework consists of contract decomposition, decomposition verification, and subsystem decomposition, iterating through contract decomposition and verification processes. Contract decomposition is carried out either through manual subsystem design or by synthesizing components from the library based on a proposed system topology. Decomposition verification ensures that incorrect decompositions are identified and addressed. If the decomposition is incorrect, the designer can leverage the quotient operation to determine the missing components. Once a contract is decomposed, the resulting subsystems, along with their contracts, can undergo further decomposition, , leading to a recursive subsystem decomposition process that loops back to the contract decomposition stage. This iterative approach continues until the top-level specification is ultimately broken down into contracts that can either be synthesized into an implementation or directly correspond to existing implementations in the library. At the final stage, the contract-based design process concludes, and the resulting contracts are mapped to actual implementations using implementation synthesis algorithms, as discussed in Section 3.6.3, and integrated into the final system implementation.

With automation support for the contract-based design framework, the methodology's benefits, such as independent design, component reuse, early integration testing, and design space exploration, can be effectively leveraged through the provided functionality.

8.5 Practical Experience

ContractDA has been utilized within the DARPA SDCPS project [44] for UAV electrical system design. The library of components, including propellers, motors, and batteries, was formulated as constraint-behavior contracts, while the top-level goal was expressed as assume-guarantee contracts. This enabled both the verification of selected components, as discussed in Chapter 4, and synthesis from the component library, as covered in Chapter 7. The application demonstrates ContractDA's ability to provide automation support for contract-based design in practical design problems.

Furthermore, ContractDA has also been used in contract research for developing contract theories and algorithms, supporting experiments presented in Chapters 4, 5, and 6. These experiments leverage ContractDA to evaluate whether the proposed algorithms correctly achieve the required tasks. Its involvement in both design applications and research underscores ContractDA's capability to bolster contract-based design and facilitate its adoption in design tasks.

8.6 Conclusion

We have presented ContractDA, a new tool that provides comprehensive support for contract-based design automation tasks. To the best of our knowledge, it is the first tool that integrates all functionalities for contract-based design, including design tasks and contract manipulations. The tool's three-layer architecture and provided interface enable both flexibility in its use and extensibility to accommodate new formalisms and algorithms.

Future work includes extending the tool and applying it to a broader range of design applications. For tool extensions, we plan to explore more contract formalisms, integrate different solvers to support additional set expressions, such as temporal logic, and enhance the interactive shell to enable contract manipulations and set operations. Expanding its application will not only demonstrate the tool's capabilities but also facilitate the adoption of contract-based design.

Chapter 9

Conclusion and Future Work

This chapter summarizes the key contributions of this dissertation and outlines promising directions for future research in advancing design automation for contract-based design.

9.1 Conclusion

Contract-based design, combining specification, hierarchical decomposition of design problems, and formal methods, has emerged as a promising methodology for addressing CPS design challenges—modeling, specification, and integration. To ensure its effective and efficient application in design problems, design automation support is crucial for reducing human errors and accelerating the process.

This dissertation advances the state of the art in design automation for contract-based design through theories, algorithms, methodologies, and tool development. Theories establish a solid foundation by enabling precise and compact specifications of physical components and ensuring correct decomposition. Algorithms, built upon these theories, define procedures for key contract-based design tasks such as verification, synthesis, and simulation, using fundamental set operations and solvers for reasoning about sets. Methodologies guide decision-making and the sequencing of design tasks to maximize efficiency and optimize performance. Finally, tool development provides an interface for applying contract-based design to real-world problems and supports research aimed at further advancing these aspects.

In Chapter 3, we examined state-of-the-art algorithms and tools for contract-based design. The key tasks, including specification, verification, simulation, and synthesis, are identified in analogy to the classic design automation paradigm widely adopted in electronic design automation. For each category, we described the general problem formulation, existing algorithms for solving it, and the available tool support. The examination result reveals gaps in theories and algorithms for specifying physical components, ensuring correct decomposition, and verifying the alignment of a contract with the design intent. These gaps motivated our research and present opportunities to advance design automation in contract-based design.

In the subsequent chapters, we analyzed these gaps and proposed solutions. In Chapter 4,

we addressed the challenge of specifying physical components by introducing *constraint-behavior contracts*, a new contract formalism. A key limitation of assume-guarantee contracts is that their properties of *implicit port directions*, which prevents them from handling environments where some controlled variables are absent from the assumption. Since these environments are inherently treated as specification violations, designers must explicitly define multiple contract versions for different port direction combinations and express behaviors through explicit equations, both of which are cumbersome for modeling physical components. Constraint-behavior contracts resolve this issue by simultaneously capturing a component's capabilities and enforcing the constraints that identify invalid environments. By deriving contract operations and transformations between the two formalisms, constraint-behavior contracts seamlessly integrate into existing contract-based design flows while enhancing expressiveness and usability for physical components.

In Chapter 5, we identified the conditions necessary for ensuring correct contract decomposition, a crucial verification step in contract-based design that guarantees correct-by-construction implementation and enables early integration testing. We observed that relying solely on refinement does not ensure correctness, as *vacuous implementations*, those that lack valid behaviors in all environments, can arise in contract decomposition due to conflicts between decomposed subsystems or with the environment. To address this issue, we first analyzed the root causes of vacuous implementations and defined *strong replaceability* as the key requirement for avoiding them. We then explored contract theory to determine the conditions contracts must satisfy to ensure strong replaceability. Our findings reveal that *receptiveness* guarantees strong replaceability in single-contract refinement and cascade composition. However, feedback composition presents additional challenges, as the behavior of one component can influence others. To tackle this, we encoded behavioral dependencies into *obligation graphs* and established that the guarantee of strong replaceability can be determined through graph properties, such as the presence of cycles, and infinite paths. These insights inspired the development of SMT-based algorithms for detecting incorrect contract decomposition by encoding these graph properties into formal constraints. With the theories and algorithms proposed in this chapter, verification can ensure correct contract decomposition, and thus enhancing reliability and robustness of contract-based design methodologies.

In Chapter 6, we introduced simulation into contract-based design to examine the alignment between design intent and contracts. The simulation is an aspect not yet explored in the literature but crucial for ensuring the correctness of contract specification. The notions of *critical behavior collections* and *critical component collections* facilitate this examination by enabling the comparison of key behaviors and identifying potential mistakes in contract expressions. The proposed simulation framework, combined with the concept of environment constraints, supports various simulation scenarios, including additional connections, specific input conditions, and computed port value evaluations. Together, these techniques provide a novel approach for validating design intent and correcting errors introduced during contract formulation.

In Chapter 7, we advanced contract-based component selection, one of the most impor-

tant synthesis problems, by enabling behavioral considerations. The inclusion of behaviors expands the applicability of contract-based synthesis to problems where behaviors, or equivalently, port values, serve as optimization objectives. We proposed an algorithm that combines black-box optimization with contract-based system reasoning to evaluate behaviors, verify refinement, and identify an initial feasible solution. This approach eliminates the need for manual formulation and efficiently handles objectives involving port values, as demonstrated in our experiments on a UAV electrical system design problem.

Consequently, the theories, algorithms, and methodologies across specification, verification, simulation, and synthesis have outlined a blueprint for automating contract-based design while ensuring the correctness of the final implementation.

Finally, we developed ContractDA, a contract-based design automation tool that integrates our proposed algorithms and essential contract manipulations to support both designers applying contract-based design and researchers exploring contract theories. As presented in Chapter 8, the tool provides multiple interfaces, including a Python API, an interactive shell, and command scripts, accommodating various levels of engagement with contract-based design. Its three-layer abstraction that separates sets, contracts, and systems offers a clear division of contract-based design tasks, manipulations, and foundational concepts, enabling extensibility at the appropriate abstraction level. By leveraging automation, ContractDA facilitates the adoption of contract-based design while ensuring correct-by-construction results.

9.2 Future work

To facilitate the adoption of contract-based design as presented in this thesis, we categorize future research directions into four major areas: theory, algorithms, tools, and applications.

9.2.1 Theory

Our work focused on developing a theoretical foundation for general set operations applicable to all contracts and set expressions. While this generality provides a universal approach, specific contract formalisms and expressions, due to their unique properties, may enable more efficient solutions tailored to their characteristics. Future research is needed to explore specialized techniques that leverage domain-specific properties to improve computational efficiency. These advancements will enhance both the applicability and practicality of contract-based design in real-world scenarios.

9.2.1.1 Reasoning in Various Set Expressions

Set operations and reasoning serve as fundamental support for contract-based design. Enhancing these operations for various set expressions and integrating advanced solvers can further expand the methodology's applicability. In this thesis, we demonstrated their use

in first-order logic. Beyond first-order logic, advanced logics such as LTL [137] and signal temporal logic (STL) [104] have been applied to contract-based design [75, 121]. Exploring theoretical foundations and enabling automated reasoning in these logics, such as formulating and solving SMT problems for decomposition verification, will be essential for supporting applications that rely on them.

9.2.1.2 Stability in Contracts

In contract-based design, the behaviors allowed by composed contracts do not necessarily ensure stability or reachability through subsystem interactions. This limitation arises from the nondeterministic semantics of composition [105], which require behaviors only to satisfy the constraints specified by the contracts, without considering their stability or reachability. As a result, contract-based design may permit implementations—particularly those involving feedback composition—that lack stable behaviors. Unstable behaviors can deviate under even small perturbations and are therefore undesirable in real-world scenarios, where physical quantities inevitably fluctuate due to noise. For example, we can arbitrarily manipulate the values of A_{OL} and β in Example 5.4 without noticing any issues in the contract operations, even when the open-loop gain violates the Nyquist criterion [128], indicating that the feedback amplifier is unstable. Incorporating constructive fixed-point semantics [52, 160] into contract theories could ensure that only stable behaviors are analyzed, addressing this limitation. Exploring this integration presents a promising direction for expanding the methodology's applicability.

9.2.2 Algorithms

Advancements in algorithms enhance both the efficiency and quality of design, expanding the scalability of methodologies. The theories discussed above should be leveraged to develop new algorithms that further strengthen contract-based design. Additionally, contract synthesis remains a challenging problem due to its inherent complexity. Promising research directions in this area include topology synthesis and learning-based synthesis, which can help automate and optimize the contract generation process, making contract-based design more practical and scalable.

9.2.2.1 Automatic Topology Synthesis

In addition to contract selection, system-level connections significantly impact overall system behavior. Therefore, system topologies should be integrated into the synthesis flow, particularly in component selection problems. Existing methods, such as those by Iannopollo *et al.* [73] and Xiao *et al.* [179], typically rely on parameter-based objective functions and discrete optimization tools to explore design configurations. However, as demonstrated in this thesis, this approach is inefficient for general component selection problems that involve objectives specified by behaviors. To address this limitation, new methods for exploring

topology during component selection must be developed and incorporated into the synthesis flow, enabling more efficient and scalable contract-based design.

9.2.2.2 Learning-based Synthesis

Since component reuse is a fundamental advantage of contract-based design, it is reasonable to assume that component libraries can be shared across different design problems. Consequently, preprocessing the library or leveraging past design experiences can significantly enhance efficiency when reusing existing designs. For instance, Iannopollo *et al.* [75] proposed an approach that pre-processes contract libraries to eliminate unnecessary refinement searches. Building on this idea, preprocessing and learning techniques can be employed to identify incompatible contracts, recognize superior partial designs that optimize design objectives, and prioritize promising exploration paths. Rule-based preprocessing methods and reinforcement learning techniques [114, 166] could further enhance automated synthesis by guiding the design space exploration more effectively. We envision that contract-based synthesis could achieve both improved efficiency and higher-quality design outcomes by incorporating these learning-driven strategies.

9.2.3 Tools

Continual tool development ensures that the advancements in research can be leveraged as a foundation for further applications and studies.

9.2.3.1 Tool Extensions

Directions for tool extensions include contract formalisms, algorithms, and integrations with applications. Extensions to contract formalisms can leverage variants such as hypercontracts [80], information flow contracts [14], and stochastic contracts [100]. Incorporating these extensions will enable to develop a broader range of applications based on the existing contract formalisms. Extensions to algorithms can enhance the tool's capability to assist designers throughout the design process. These extensions should include both existing algorithms and newly developed ones, such as those mentioned above. Relevant existing algorithms include contract refinement tightening [32, 33] and internal port elimination [79]. Incorporating these algorithms will expand the tool's functionality, improving efficiency and flexibility for designers.

Integration with specific applications is also crucial, even though it is not the primary focus of contract-based design. Some applications allow components to be universally expressive, meaning that implementations can be derived for any expression within the domain. For example, standard cells in digital integrated circuits can implement any Boolean formula expressions. In such cases, the resulting contracts should be structured to support implementation synthesis for the targeted application. To facilitate this process, a dedicated interface and file format should be developed for integration with industry standards, reduc-

ing the designer's burden in format conversion. For instance, in digital integrated circuits, contracts must be translated into RTL descriptions, such as Verilog or SystemVerilog, to enable domain-specific synthesis steps.

9.2.3.2 Combination with Large Language Model

Large Language Models (LLMs) have emerged as powerful tools for reasoning and generating natural language, as well as programming languages [4, 63, 183]. Various models and applications have been developed to assist designers in tasks such as algorithm writing, reasoning about complex mathematical problems, and generating or refining articles. This breakthrough in artificial intelligence opens new opportunities for contract-based design, including LLM-assisted contract writing and contract-assisted LLM-based design.

Specification is a crucial challenge in CPS design, and converting design intent—especially when expressed in natural language—into contracts remains a major obstacle to adopting contract-based design. If LLM-assisted contract writing can be developed, natural language design intent could be efficiently translated into contract expressions. This would significantly reduce the effort required for designers to learn contract formalisms and manually formulate contracts from scratch. Additionally, LLMs could potentially assist in synthesis by proposing contract decompositions. In short, LLMs are promising in reducing human intervention in contract-based design tasks, making contract-based design more accessible and efficient.

On the other hand, correctness remains crucial in contract-based design. This must be ensured through the use of well-defined contract decomposition concepts and the alignment of contract semantics with design intent. This can lead to the concept of contract-assisted LLM-based design, where contract verification and simulation are applied to validate the correctness of proposed contract formulations and decompositions. We envision that integrating formal methods with generative models will be a key approach to leveraging the power of artificial intelligence while maintaining correctness in design outcomes.

9.2.4 Applications

This dissertation has focused on design automation of contract-based design for cyber-physical systems. However, the design methodology can be applied to any domain that requires correct-by-construction design and decomposition to improve efficiency while maintaining flexibility in component expressions. For example, previous publications have applied contract-based design to analog circuits [126] and arithmetic logic units [81]. Expanding this methodology to various applications, such as chiplet design, mixed-signal integrated circuits, smart buildings, robotics, and network-controlled systems, is a promising direction for achieving efficient and high-quality designs.

To enable contract-based design in an application domain, a well-defined component library and a behavior modeling framework based on set operations are essential. These elements form a foundation for designers to systematically apply contract-based methodologies.

With a well-defined component library, a design goal can be decomposed into subsystems from available components, leading to an implementable solution. The behavior modeling framework enables defining the design goals and the component library at an appropriate level of abstraction. Collaboration with domain experts is crucial to ensure that the component library accurately captures all relevant behavioral aspects and that the modeling framework provides sufficient abstraction to reduce design complexity while remaining expressive enough to define design goals encompassing all key design aspects. For example, in chiplet design, modeling and expressing component behaviors, including UCIe links, electrical and optical connections, and computing capabilities, are necessary to ensure that the system can support a target application workload while optimizing area, latency, and power consumption.

Bibliography

[1] Martin Abadi and Leslie Lamport. "Composing specifications". In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 15.1 (1993), pp. 73–132.

[2] Martin Abadi and Leslie Lamport. "Conjoining specifications". In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 17.3 (1995), pp. 507–535.

[3] Martin Abadi, Leslie Lamport, and Pierre Wolper. "Realizable and unrealizable specifications of reactive systems". In: *International Colloquium on Automata, Languages, and Programming*. 1989, pp. 1–17.

[4] Josh Achiam et al. "Gpt-4 technical report". In: *arXiv preprint arXiv:2303.08774* (2023).

[5] Takuya Akiba et al. "Optuna: A Next-generation Hyperparameter Optimization Framework". In: *Proc. of KDD*. 2019, pp. 2623–2631.

[6] Luca de Alfaro and Thomas A. Henzinger. "Interface Automata". In: *Proceedings of the 8th European Software Engineering Conference*. New York, NY, USA: Association for Computing Machinery, 2001, pp. 109–120.

[7] Rajeev Alur et al. "Alternating refinement relations". In: *CONCUR'98 Concurrency Theory: 9th International Conference*. Springer. 1998, pp. 163–178.

[8] Roy Armoni et al. "Enhanced vacuity detection in linear temporal logic". In: *Computer Aided Verification*. Springer. 2003, pp. 368–380.

[9] Autodesk. *AutoCAD*. Available: `https://www.autodesk.com/products/autocad/overview` Accessed: 2025-03-20.

[10] Ralph-Johan Back and Joakim von Wright. "Contracts, games, and refinement". In: *Information and Computation* 156.1-2 (2000), pp. 25–45.

[11] Ralph-Johan Back and Joakim von Wright. *Refinement calculus: a systematic introduction*. Springer Science & Business Media, 2012.

[12] Clark Barrett and Cesare Tinelli. *Satisfiability modulo theories*. Springer, 2018.

[13] Clark Barrett et al. "CVC4". In: *Proceedings of the 23rd International Conference on Computer Aided Verification*. 2011, pp. 171–177.

[14] Ezio Bartocci et al. "Information-flow Interfaces". In: *Fundamental Approaches to Software Engineering*. Springer International Publishing, 2022, pp. 3–22.

[15] Sebastian S Bauer et al. "Moving from specifications to contracts in component-based design". In: *Fundamental Approaches to Software Engineering: 15th International Conference*. Springer. 2012, pp. 43–58.

[16] Albert Benveniste et al. *Contracts for system design*. Tech. rep. Inria Rennes Bretagne Atlantique; INRIA, 2012.

[17] Albert Benveniste et al. "Contracts for system design". In: *Foundations and Trends® in Electronic Design Automation* 12.2-3 (2018), pp. 124–400.

[18] Albert Benveniste et al. *Contracts for systems design: methodology and application cases*. Tech. rep. Inria Rennes Bretagne Atlantique; INRIA, 2015.

[19] Albert Benveniste et al. *Contracts for systems design: Theory*. Tech. rep. Inria Rennes Bretagne Atlantique; INRIA, 2015.

[20] Albert Benveniste et al. "Multiple viewpoint contract-based specification and design". In: *International Symposium on Formal Methods for Components and Objects*. 2007, pp. 200–225.

[21] Luca Benvenuti et al. "A contract-based formalism for the specification of heterogeneous systems". In: *2008 Forum on Specification, Verification and Design Languages*. IEEE. 2008, pp. 142–147.

[22] Luca Benvenuti et al. "Contract-based design for computation and verification of a closed-loop hybrid system". In: *International Workshop on Hybrid Systems: Computation and Control*. Springer. 2008, pp. 58–71.

[23] James Bergstra et al. "Algorithms for hyper-parameter optimization". In: *Proc. of NeurIPS* (2011), pp. 2546–2554.

[24] Roderick Bloem et al. "RATSY–a new requirements analysis tool with synthesis". In: *Computer Aided Verification: 22nd International Conference*. Springer. 2010, pp. 425–429.

[25] Roderick Bloem et al. "Synthesizing robust systems". In: *Acta Informatica* 51 (2014), pp. 193–220.

[26] Wim Bogaerts and Lukas Chrostowski. "Silicon photonics circuit design: methods, tools and challenges". In: *Laser & Photonics Reviews* 12.4 (2018), p. 1700237.

[27] Cadence Design Systems. *OrCAD X Platform*. Available: `https://www.cadence.com/en_US/home/tools/pcb-design-and-analysis/orcad.html` Accessed: 2025-03-20.

[28] Roberto Cavada et al. "The nuXmv symbolic model checker". In: *Computer Aided Verification: 26th International Conference*. Springer. 2014, pp. 334–342.

[29] Arindam Chakrabarti et al. "Resource interfaces". In: *International Workshop on Embedded Software*. Springer. 2003, pp. 117–133.

[30] Krishnendu Chatterjee and Thomas A Henzinger. "Assume-guarantee synthesis". In: *Tools and Algorithms for the Construction and Analysis of Systems: 13th International Conference*. Springer. 2007, pp. 261–275.

[31] Alonzo Church. "Logic, arithmetic and automata". In: *Proceedings of the international congress of mathematicians*. Vol. 1962. 1962, pp. 23–35.

[32] Alessandro Cimatti, Ramiro Demasi, and Stefano Tonetta. "Tightening a contract refinement". In: *Software Engineering and Formal Methods: 14th International Conference*. Springer. 2016, pp. 386–402.

[33] Alessandro Cimatti, Ramiro Demasi, and Stefano Tonetta. "Tightening the contract refinements of a system architecture". In: *Formal Methods in System Design* 52 (2018), pp. 88–116.

[34] Alessandro Cimatti, Michele Dorigatti, and Stefano Tonetta. "OCRA: A tool for checking the refinement of temporal contracts". In: *IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 2013, pp. 702–705.

[35] Alessandro Cimatti and Stefano Tonetta. "A property-based proof system for contract-based design". In: *Euromicro Conference on Software Engineering and Advanced Applications*. 2012, pp. 21–28.

[36] Alessandro Cimatti et al. "The mathsat5 smt solver". In: *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer. 2013, pp. 93–107.

[37] Lawrence T Clark et al. "ASAP7: A 7-nm finFET predictive process design kit". In: *Microelectronics Journal* 53 (2016), pp. 105–115.

[38] E.M. Clarke, D.E. Long, and K.L. McMillan. "Compositional model checking". In: *Proceedings of the Fourth Annual Symposium on Logic in computer science*. 1989, pp. 353–362.

[39] Jamieson M Cobleigh, George S Avrunin, and Lori A Clarke. "Breaking up is hard to do: An evaluation of automated assume-guarantee reasoning". In: *ACM Transactions on Software Engineering and Methodology (TOSEM)* 17.2 (2008), pp. 1–52.

[40] Darren Cofer et al. "Compositional Verification of Architectural Models". In: *NASA Formal Methods*. Springer Berlin Heidelberg, 2012, pp. 126–140.

[41] Andrew R Conn, Katya Scheinberg, and Luis N Vicente. *Introduction to derivative-free optimization*. SIAM, 2009.

[42] Werner Damm et al. "Boosting re-use of embedded automotive applications through rich components". In: *Proceedings of foundations of interface technologies* 2005 (2005).

[43] Werner Damm et al. "Using contract-based component specifications for virtual integration testing and architecture design". In: *Design, Automation & Test in Europe Conference Exhibition (DATE)*. 2011, pp. 1–6.

[44] DARPA. *SDCPS Project*. Available at `https://www.darpa.mil/program/symbiotic-design-for-cyber-physical-systems`.

[45] Alexandre David et al. "Methodologies for specification of real-time systems using timed I/O automata". In: *International Symposium on Formal Methods for Components and Objects*. Springer. 2009, pp. 290–310.

[46] Luca De Alfaro and Thomas A Henzinger. "Interface theories for component-based design". In: *International Workshop on Embedded Software*. Springer. 2001, pp. 148–165.

[47] Luca De Alfaro, Thomas A Henzinger, and Mariëlle Stoelinga. "Timed interfaces". In: *Embedded Software: Second International Conference*. Springer. 2002, pp. 108–122.

[48] Leonardo De Moura and Nikolaj Bjørner. "Z3: An Efficient SMT Solver". In: *International conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer. 2008, pp. 337–340.

[49] Edsger W Dijkstra. *A discipline of programming*. prentice-hall Englewood Cliffs, 1976.

[50] David L Dill. *Trace theory for automatic hierarchical verification of speed-independent circuits*. Vol. 24. MIT press, 1989.

[51] Laurent Doyen et al. "Interface theories with component reuse". In: *Proceedings of the 8th ACM international conference on Embedded software*. 2008, pp. 79–88.

[52] Stephen A Edwards and Edward A Lee. "The semantics and execution of a synchronous block-diagram language". In: *Science of Computer Programming* 48.1 (2003), pp. 21–42.

[53] Peter H Feiler, David P Gluch, and John Hudak. *The architecture analysis & design language (AADL): An introduction*. Carnegie Mellon University, Software Engineering Institute, 2006.

[54] Ioannis Filippidis. "Decomposing formal specifications into assume-guarantee contracts for hierarchical system design". PhD thesis. California Institute of Technology, 2019.

[55] John Finn, Pierluigi Nuzzo, and Alberto Sangiovanni-Vincentelli. "A mixed discrete-continuous optimization scheme for cyber-physical system architecture exploration". In: *IEEE/ACM Internation Conference on Computer-Aided Design (ICCAD)*. IEEE. 2015, pp. 216–223.

[56] Robert W. Floyd. "Assigning Meanings to Programs". In: *Program Verification: Fundamental Issues in Computer Science*. Springer Netherlands, 1993, pp. 65–81.

[57] Daniel J Fremont et al. "Scenic: a language for scenario specification and scene generation". In: *Proceedings of the 40th ACM SIGPLAN conference on programming language design and implementation*. 2019, pp. 63–78.

[58] Peter Fritzson and Vadim Engelson. "Modelica-a unified object-oriented language for system modeling and simulation". In: *ECOOP*. Vol. 98. Citeseer. 1998, pp. 67–90.

[59] Kasra Ghasemi, Sadra Sadraddini, and Calin Belta. "Compositional synthesis via a convex parameterization of assume-guarantee contracts". In: *Proceedings of the 23rd International Conference on Hybrid Systems: Computation and Control*. 2020, pp. 1–10.

[60] Gregor Goessler and Jean-Baptiste Raclet. "Modal contracts for component-based design". In: *2009 Seventh IEEE International Conference on Software Engineering and Formal Methods*. IEEE. 2009, pp. 295–303.

[61] Josefine B Graebener, Apurva Badithela, and Richard M Murray. "Towards better test coverage: Merging unit tests for autonomous systems". In: *NASA Formal Methods Symposium*. Springer. 2022, pp. 133–155.

[62] Orna Grumberg and David E Long. "Model checking and modular verification". In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 16.3 (1994), pp. 843–871.

[63] Daya Guo et al. "Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning". In: *arXiv preprint arXiv:2501.12948* (2025).

[64] David Harel and Amir Pnueli. "On the development of reactive systems". In: *Logics and models of concurrent systems*. Springer, 1984, pp. 477–498.

[65] Thomas A Henzinger, Ranjit Jhala, and Rupak Majumdar. "Permissive interfaces". In: *Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering*. 2005, pp. 31–40.

[66] Carl Hewitt, Peter Bishop, and Richard Steiger. "SA Universal Modular Actor Formalism for Artificial Intelligence". In: *International Joint Conferences on Artificial Intelligence*. Vol. 3. 1973, p. 235.

[67] Charles Antony Richard Hoare. "An axiomatic basis for computer programming". In: *Communications of the ACM* 12.10 (1969), pp. 576–580.

[68] Charles Antony Richard Hoare et al. *Communicating sequential processes*. Vol. 178. Prentice-hall Englewood Cliffs, 1985.

[69] Gerard J. Holzmann. "The model checker SPIN". In: *IEEE Transactions on software engineering* 23.5 (1997), pp. 279–295.

[70] Xing Huang et al. "Computer-aided design techniques for flow-based microfluidic lab-on-a-chip systems". In: *ACM Computing Surveys (CSUR)* 54.5 (2021), pp. 1–29.

[71] Antonio Iannopollo. "A Platform-Based Approach to Verification and Synthesis of Linear Temporal Logic Specifications". PhD thesis. University of California, Berkeley, 2018.

[72] Antonio Iannopollo, Inigo Incer, and Alberto L Sangiovanni-Vincentelli. "Synthesizing LTL contracts from component libraries using rich counterexamples". In: *Science of Computer Programming* (2024), pp. 103–116.

[73] Antonio Iannopollo, Stavros Tripakis, and Alberto Sangiovanni-Vincentelli. "Constrained Synthesis from Component Libraries". In: *Formal Aspects of Component Software*. 2017, pp. 92–110.

[74] Antonio Iannopollo, Stavros Tripakis, and Alberto Sangiovanni-Vincentelli. "Constrained synthesis from component libraries". In: *Science of Computer Programming* 171 (2019), pp. 21–41.

[75] Antonio Iannopollo, Stavros Tripakis, and Alberto Sangiovanni-Vincentelli. "Specification decomposition for synthesis from libraries of LTL Assume/Guarantee contracts". In: *Design, Automation & Test in Europe Conference Exhibition (DATE)*. 2018, pp. 1574–1579.

[76] Antonio Iannopollo et al. "Library-based scalable refinement checking for contract-based design". In: *Design, Automation & Test in Europe Conference Exhibition (DATE)*. IEEE. 2014, pp. 1–6.

[77] Inigo Incer. "The Algebra of Contracts". PhD thesis. EECS Department, University of California, Berkeley, May 2022.

[78] Inigo Incer et al. "Hypercontracts". In: *NASA Formal Methods*. 2022, pp. 674–692.

[79] Inigo Incer et al. "Pacti: Assume-Guarantee Contracts for Efficient Compositional Analysis and Design". In: *ACM Trans. Cyber-Phys. Syst.* (2024).

[80] Inigo Incer et al. "Pacti: Scaling assume-guarantee reasoning for system analysis and design". In: *arXiv preprint arXiv:2303.17751* (2023).

[81] Inigo Incer et al. "Quotient for assume-guarantee contracts". In: *2018 16th ACM/IEEE International Conference on Formal Methods and Models for System Design (MEMOCODE)*. IEEE. 2018, pp. 1–11.

[82] Barbara Jobstmann et al. "Anzu: A Tool for Property Synthesis: (Tool Paper)". In: *Computer Aided Verification: 19th International Conference*. Springer. 2007, pp. 258–262.

[83] C. B. Jones. "Tentative Steps toward a Development Method for Interfering Programs". In: *ACM Transactions on Programming Languages and Systems* 5.4 (1983), pp. 596–619.

[84] Gilles Kahn. "The semantics of a simple language for parallel programming." In: *IFIP congress*. Vol. 74. 1974, pp. 471–475.

[85] Gabor Karsai et al. "Model-integrated development of embedded software". In: *Proceedings of the IEEE* 91.1 (2003), pp. 145–164.

[86] Andreas Katis et al. "Validity-guided synthesis of reactive systems from assume-guarantee contracts". In: *Tools and Algorithms for the Construction and Analysis of Systems: 24th International Conference*. Springer. 2018, pp. 176–193.

[87] Kurt Keutzer et al. "System-level design: Orthogonalization of concerns and platform-based design". In: *IEEE transactions on computer-aided design of integrated circuits and systems* 19.12 (2000), pp. 1523–1543.

[88] Dmitrii Kirov et al. "ArchEx: An extensible framework for the exploration of cyber-physical system architectures". In: *IEEE/ACM Design Automation Conference (DAC)*. 2017, pp. 1–6.

[89] Dmitrii Kirov et al. "Optimized selection of wireless network topologies and components via efficient pruning of feasible paths". In: *IEEE/ACM Design Automation Conference (DAC)*. 2018, pp. 1–6.

[90] Marius Kloetzer and Calin Belta. "A fully automated framework for control of linear systems from temporal logic specifications". In: *IEEE Transactions on Automatic Control* 53.1 (2008), pp. 287–297.

[91] Hadas Kress-Gazit, Georgios E Fainekos, and George J Pappas. "Temporal-logic-based reactive mission and motion planning". In: *IEEE transactions on robotics* 25.6 (2009), pp. 1370–1381.

[92] Leslie Lamport. "Specifying concurrent program modules". In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 5.2 (1983), pp. 190–222.

[93] Leslie Lamport. "Specifying systems: the TLA+ language and tools for hardware and software engineers". In: (2002).

[94] Kim G Larsen, Ulrik Nyman, and Andrzej Wąsowski. "Interface input/output automata". In: *International Symposium on Formal Methods*. Springer. 2006, pp. 82–97.

[95] Kim G Larsen, Ulrik Nyman, and Andrzej Wąsowski. "Modal I/O automata for interface and product line theories". In: *Programming Languages and Systems: 16th European Symposium on Programming*. Springer. 2007, pp. 64–79.

[96] Thi Thieu Hoa Le et al. "Contract-Based Requirement Modularization via Synthesis of Correct Decompositions". In: *ACM Transactions on Embedded Computing Systems (TECS)* 15.2 (2016).

[97] Ákos Lédeczi et al. "Composing domain-specific design environments". In: *Computer* 34.11 (2001), pp. 44–51.

[98] Edward A Lee and Alberto Sangiovanni-Vincentelli. "A framework for comparing models of computation". In: *IEEE Transactions on computer-aided design of integrated circuits and systems* 17.12 (1998), pp. 1217–1229.

[99] Edward Ashford Lee and Sanjit Arunkumar Seshia. *Introduction to embedded systems: A cyber-physical systems approach*. MIT Press, 2016.

[100] Jiwei Li et al. "Stochastic contracts for cyber-physical system design under probabilistic requirements". In: *Proceedings of the 15th ACM-IEEE International Conference on Formal Methods and Models for System Design*. 2017, pp. 5–14.

[101] Nancy A. Lynch and Mark R. Tuttle. "Hierarchical Correctness Proofs for Distributed Algorithms". In: *Proceedings of the Sixth Annual ACM Symposium on Principles of Distributed Computing*. Association for Computing Machinery, 1987, pp. 137–151.

[102] Mehdi Maasoumy, Pierluigi Nuzzo, and Alberto Sangiovanni-Vincentelli. *Smart buildings in the smart grid: Contract-based design of an integrated energy management system*. Springer, 2015.

[103] Rupak Majumdar et al. "Assume–Guarantee Distributed Synthesis". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)* 39.11 (2020), pp. 3215–3226.

[104] Oded Maler and Dejan Nickovic. "Monitoring temporal properties of continuous signals". In: *International Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems*. Springer. 2004, pp. 152–166.

[105] Sharad Malik. "Analysis of cyclic combinational circuits". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 13.7 (1994), pp. 950–956.

[106] Piergiuseppe Mallozzi et al. "Crome: contract-based robotic mission specification". In: *2020 18th ACM-IEEE International Conference on Formal Methods and Models for System Design (MEMOCODE)*. IEEE. 2020, pp. 1–11.

[107] Zohar Manna and Richard Waldinger. "A deductive approach to program synthesis". In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 2.1 (1980), pp. 90–121.

[108] Michael C McCord, J William Murdock, and Branimir K Boguraev. "Deep parsing in Watson". In: *IBM Journal of research and development* 56.3.4 (2012), pp. 3–1.

[109] Bertrand Meyer. "Applying 'design by contract'". In: *Computer* 25.10 (1992), pp. 40–51.

[110] Bertrand Meyer. "Touch of class". In: *Learning to program well with Object Technology and Design by Contract, AN INTRODUCTION TO SOFTWARE ENGINEERING http://se. inf. ethz. ch/touch* (2009), p. 51.

[111] Bertrand Meyer. "Toward More expressive contracts". In: *Journal of Object Oriented Programming* 13.4 (2000), pp. 39–43.

[112] Alan Mishchenko et al. *FRAIGs: A unifying representation for logic synthesis and verification*. Tech. rep. ERL Technical Report, 2005.

[113] Ashish Mishra and Suresh Jagannathan. "Specification-guided component-based synthesis from effectful libraries". In: *Proceedings of the ACM on Programming Languages* (2022), pp. 616–645.

[114] Volodymyr Mnih et al. "Human-level control through deep reinforcement learning". In: *nature* 518.7540 (2015), pp. 529–533.

[115] James Donald Monk. *Mathematical logic*. Vol. 37. Springer Science & Business Media, 2012.

[116] Andreas Müller et al. "Tactical contract composition for hybrid system component verification". In: *International Journal on Software Tools for Technology Transfer* 20 (2018), pp. 615–643.

[117] Radu Negulescu. "Process spaces". In: *International Conference on Concurrency Theory*. Springer. 2000, pp. 199–213.

[118] Gabriela Nicolescu and Pieter J Mosterman. *Model-based design for embedded systems*. Crc Press, 2018.

[119] Pierluigi Nuzzo. "From Electronic Design Automation to Cyber-Physical System Design Automation: A Tale of Platforms and Contracts". In: *Proceedings of the International Symposium on Physical Design (ISPD)*. San Francisco, CA, USA, 2019, pp. 117–121.

[120] Pierluigi Nuzzo and Alberto L. Sangiovanni-Vincentelli. "Hierarchical System Design with Vertical Contracts". In: *Principles of Modeling: Essays Dedicated to Edward A. Lee on the Occasion of His 60th Birthday*. Springer International Publishing, 2018, pp. 360–382.

[121] Pierluigi Nuzzo et al. "A contract-based methodology for aircraft electric power system design". In: *IEEE Access* 2 (2013), pp. 1–25.

[122] Pierluigi Nuzzo et al. "A platform-based design methodology with contracts and related tools for the design of cyber-physical systems". In: *Proceedings of the IEEE* 103.11 (2015), pp. 2104–2132.

[123] Pierluigi Nuzzo et al. "Are interface theories equivalent to contract theories?" In: *2014 Twelfth ACM/IEEE Conference on Formal Methods and Models for Codesign (MEMOCODE)*. IEEE. 2014, pp. 104–113.

[124] Pierluigi Nuzzo et al. "CHASE: Contract-based requirement engineering for cyber-physical system design". In: *Design, Automation & Test in Europe Conference Exhibition (DATE)*. 2018, pp. 839–844.

[125] Pierluigi Nuzzo et al. "Contract-based design of control protocols for safety-critical cyber-physical systems". In: *2014 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE. 2014, pp. 1–4.

[126] Pierluigi Nuzzo et al. "Methodology for the Design of Analog Integrated Interfaces Using Contracts". In: *IEEE Sensors Journal* 12.12 (2012), pp. 3329–3345.

[127] Pierluigi Nuzzo et al. "Stochastic assume-guarantee contracts for cyber-physical system design". In: *ACM Transactions on Embedded Computing Systems (TECS)* 18.1 (2019), pp. 1–26.

[128] Harry Nyquist. "Regeneration theory". In: *Bell system technical journal* 11.1 (1932), pp. 126–147.

[129] Chanwook Oh, Michele Lora, and Pierluigi Nuzzo. "Quantitative Verification and Design Space Exploration Under Uncertainty with Parametric Stochastic Contracts". In: *IEEE/ACM Internation Conference on Computer-Aided Design (ICCAD)*. 2022, pp. 1–9.

[130] Chanwook Oh et al. "Optimizing assume-guarantee contracts for cyber-physical system design". In: *Design, Automation & Test in Europe Conference Exhibition (DATE)*. 2019, pp. 246–251.

[131] *OMG Systems Modeling Language (SysML)*. Object Management Group (OMG), 2024. URL: `https://www.omg.org/spec/SysML/`.

[132] Roberto Passerone, Inigo Incer, and Alberto L Sangiovanni-Vincentelli. "Coherent Extension, Composition, and Merging Operators in Contract Models for System Design". In: *ACM Transactions on Embedded Computing Systems (TECS)* 18.5 (2019), pp. 1–23.

[133] Steffen Peter and Tony Givargis. "Component-based synthesis of embedded systems using satisfiability modulo theories". In: *ACM Transactions on Design Automation of Electronic Systems (TODAES)* 20.4 (2015), pp. 1–27.

[134] Tung Phan-Minh. *Contract-based design: Theories and applications*. California Institute of Technology, 2021.

[135] Nir Piterman, Amir Pnueli, and Yaniv Sa'ar. "Synthesis of reactive (1) designs". In: *Verification, Model Checking, and Abstract Interpretation: 7th International Conference*. Springer. 2006, pp. 364–380.

[136] Amir Pnueli. "In transition from global to modular temporal reasoning about programs". In: *Logics and models of concurrent systems*. Springer, 1984, pp. 123–144.

[137] Amir Pnueli. "The temporal logic of programs". In: *18th annual symposium on foundations of computer science (sfcs 1977)*. ieee. 1977, pp. 46–57.

[138] Amir Pnueli. "The temporal semantics of concurrent programs". In: *Theoretical computer science* 13.1 (1981), pp. 45–60.

[139] Amir Pnueli and Roni Rosner. "On the synthesis of a reactive module". In: *Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 1989, pp. 179–190.

[140] Amir Pnueli, Yaniv Sa'ar, and Lenore D Zuck. "JTLV: A framework for developing verification algorithms". In: *International Conference on Computer Aided Verification*. Springer. 2010, pp. 171–174.

[141] PTC. *Creo*. Available: `https://www.ptc.com/en/products/creo` Accessed: 2025-03-20.

[142] Sophie Quinton and Susanne Graf. "Contract-based verification of hierarchical systems of components". In: *IEEE International Conference on Software Engineering and Formal Methods*. 2008, pp. 377–381.

[143] Jean-Baptiste Raclet et al. "A modal interface theory for component-based design". In: *Fundamenta Informaticae* 108.1-2 (2011), pp. 119–149.

[144] Jean-Baptiste Raclet et al. "Modal interfaces: unifying interface automata and modal specifications". In: *Proceedings of the seventh ACM international conference on Embedded software*. 2009, pp. 87–96.

[145] Vasumathi Raman et al. "Reactive synthesis from signal temporal logic specifications". In: *Proceedings of the 18th international conference on hybrid systems: Computation and control*. 2015, pp. 239–248.

[146] Íñigo X Íncer Romeo et al. "The quotient in preorder theories". In: *Proceedings 11th International Symposium on Games, Automata, Logics, and Formal Verification*. 2020, pp. 216–233.

[147] Kenneth H Rosen and Kamala Krithivasan. *Discrete mathematics and its applications*. Vol. 6. McGraw-hill New York, 1999.

[148] Nicolas Rouquette, Inigo Incer, and Alessandro Pinto. "Early Design Exploration of Space System Scenarios Using Assume-Guarantee Contracts". In: *2023 IEEE 9th International Conference on Space Mission Challenges for Information Technology (SMC-IT)*. IEEE. 2023, pp. 15–24.

[149] Alberto Sangiovanni-Vincentelli. "Quo vadis, SLD? Reasoning about the trends and challenges of system level design". In: *Proceedings of the IEEE* 95.3 (2007), pp. 467–506.

[150] Alberto Sangiovanni-Vincentelli. "The tides of EDA". In: *IEEE Design & Test of Computers* 20.6 (2003), pp. 59–75.

[151] Alberto Sangiovanni-Vincentelli, Werner Damm, and Roberto Passerone. "Taming Dr. Frankenstein: Contract-based design for cyber-physical systems". In: *European journal of control* 18.3 (2012), pp. 217–238.

[152] César Augusto R dos Santos et al. "Divide et Impera: Efficient Synthesis of Cyber-Physical System Architectures from Formal Contracts". In: *Formal Methods: 24th International Symposium, FM 2021, Virtual Event, November 20–26, 2021, Proceedings 24*. Springer. 2021, pp. 776–787.

[153] César Augusto Ribeiro dos Santos et al. "CONDEnSe: contract based design synthesis". In: *2019 ACM/IEEE 22nd International Conference on Model Driven Engineering Languages and Systems (MODELS)*. IEEE. 2019, pp. 250–260.

[154] Douglas C Schmidt et al. "Model-driven engineering". In: *Computer* 39.2 (2006), p. 25.

[155] A Sedra et al. "Microelectronic circuits 8th edition". In: *Chapter* 14 (2020), pp. 1235–1236.

[156] Bran Selic. "The pragmatics of model-driven development". In: *IEEE software* 20.5 (2003), pp. 19–25.

[157] Sanjit A Seshia and Pramod Subramanyan. "UCLID5: Integrating modeling, verification, synthesis and learning". In: *2018 16th ACM/IEEE International Conference on Formal Methods and Models for System Design (MEMOCODE)*. IEEE. 2018, pp. 1–10.

[158] Sanjit A. Seshia et al. "Design Automation of Cyber-Physical Systems: Challenges, Advances, and Opportunities". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)* 36.9 (2017), pp. 1421–1434.

[159] Debendra Das Sharma et al. "Universal chiplet interconnect express (UCIe): An open industry standard for innovations with chiplets at package level". In: *IEEE Transactions on Components, Packaging and Manufacturing Technology* 12.9 (2022), pp. 1423–1431.

[160] Thomas R Shiple, Gérard Berry, and Hervé Touati. "Constructive analysis of cyclic circuits". In: *Proceedings ED&TC European Design and Test Conference*. IEEE. 1996, pp. 328–333.

[161] Michael Sievers and Azad M. Madni. "A flexible contracts approach to system resiliency". In: *2014 IEEE International Conference on Systems, Man, and Cybernetics (SMC)*. 2014, pp. 1002–1007.

[162] Jonathan Slenders. *prompt_toolkit: A library for building powerful interactive command lines in Python*. Version 3.0. Accessed: 2025-05-07. 2020. URL: `https://github.com/prompt-toolkit/python-prompt-toolkit`.

[163] Mathias Soeken, Thomas Haener, and Martin Roetteler. "Programming quantum computers using design automation". In: *2018 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE. 2018, pp. 137–146.

[164] Stefano Spellini et al. "Virtual Prototyping a Production Line Using Assume–Guarantee Contracts". In: *IEEE Transactions on Industrial Informatics* 17.9 (2021), pp. 6294–6302.

[165] Minghui Sun et al. "Correct-by-construction: a contract-based semi-automated requirement decomposition process". In: *arXiv preprint arXiv:1909.02070* (2019).

[166] Richard S Sutton, Andrew G Barto, et al. *Reinforcement learning: An introduction*. Vol. 1. 1. MIT press Cambridge, 1998.

[167] Janos Sztipanovits and Gabor Karsai. "Model-integrated computing". In: *Computer* 30.4 (1997), pp. 110–111.

[168] The Mathwork Inc. *Simulink*. Available at `https://www.mathworks.com/products/simulink.html`.

[169] Stavros Tripakis et al. "A theory of synchronous relational interfaces". In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 33.4 (2011), pp. 1–41.

[170] UCIe Consortium. *UCIe Specification 2.0*. Available: `https://www.uciexpress.org/2-0-spec-download`. Aug. 2024.

[171] James D. Walker et al. "A Flight Dynamics Model for Exploring the Distributed Electrical eVTOL Cyber Physical Design Space". In: *2022 IEEE Workshop on Design Automation for CPS and IoT (DESTION)*. 2022, pp. 7–12.

[172] Laung-Terng Wang, Yao-Wen Chang, and Kwang-Ting Tim Cheng. *Electronic design automation: synthesis, verification, and test*. Morgan Kaufmann, 2009.

[173] Timothy E Wang et al. "Hierarchical contract-based synthesis for assurance cases". In: *NASA Formal Methods Symposium*. Springer. 2022, pp. 175–192.

[174] J Warmer. *The Object Constraint Language: Getting Your Models Ready for MDA*. Addison-Wesley Professional, 2003.

[175] Jonas Westman and Mattias Nyberg. "Conditions of contracts for separating responsibilities in heterogeneous systems". In: *Formal Methods in System Design* 52 (2018), pp. 147–192.

[176] Jan C. Willems. "The Behavioral Approach to Open and Interconnected Systems". In: *IEEE Control Systems Magazine* 27.6 (2007), pp. 46–99.

[177] Elizabeth Susan Wolf. *Hierarchical models of synchronous circuits for formal verification and substitution*. stanford university, 1996.

[178] Tichakorn Wongpiromsarn et al. "TuLiP: a software toolbox for receding horizon temporal logic planning". In: *Proceedings of the 14th international conference on Hybrid systems: computation and control*. 2011, pp. 313–314.

[179] Yifeng Xiao et al. "Efficient Exploration of Cyber-Physical System Architectures Using Contracts and Subgraph Isomorphism". In: *2024 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE. 2024, pp. 1–6.

[180] Sheng-Jung Yu. *ContractDA*. Available at `https://github.com/ContractDA/ContractDA`.

[181] Sheng-Jung Yu, Inigo Incer, and Alberto Sangiovanni-Vincentelli. "Constraint-Behavior Contracts: A Formalism for Specifying Physical Systems". In: *2023 21th ACM/IEEE International Conference on Formal Methods and Models for System Design (MEMOCODE)*. IEEE. 2023, pp. 1–11.

[182] Sheng-Jung Yu, Inigo Incer, and Alberto Sangiovanni-Vincentelli. "Contract Replaceability for Ensuring Independent Design using Assume-Guarantee Contracts". In: *2023 21th ACM/IEEE International Conference on Formal Methods and Models for System Design (MEMOCODE)*. IEEE. 2023, pp. 1–11.

[183] Wayne Xin Zhao et al. "A survey of large language models". In: *arXiv preprint arXiv:2303.18223* 1.2 (2023).