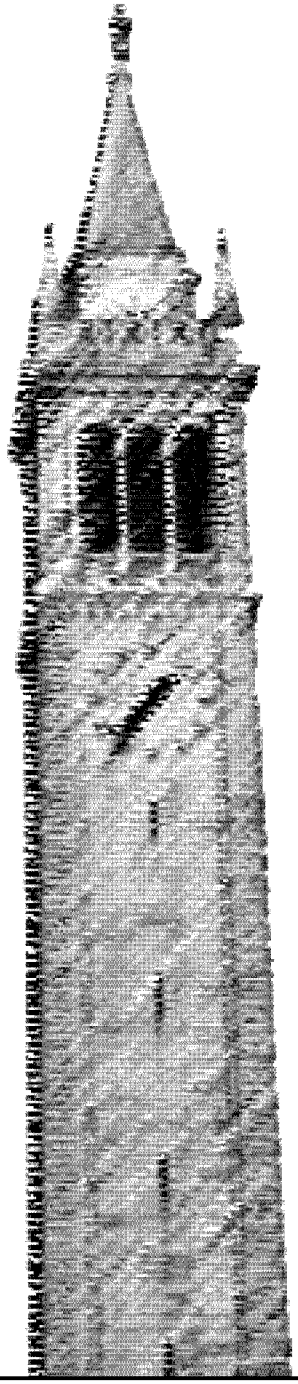
The Algebra of Contracts



Inigo Incer

Electrical Engineering and Computer Sciences
University of California, Berkeley

Technical Report No. UCB/EECS-2022-99
http://www2.eecs.berkeley.edu/Pubs/TechRpts/2022/EECS-2022-99.html

May 13, 2022

The Algebra of Contracts

by

Inigo Xabier Incer Romeo

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 Albert Benveniste
Professor Francesco Borrelli
Professor Sanjit A. Seshia

Spring 2022

The Algebra of Contracts

Abstract

The Algebra of Contracts

by

Inigo Xabier Incer Romeo

Doctor of Philosophy in Engineering - Electrical Engineering and Computer Sciences

University of California, Berkeley

Professor Alberto Sangiovanni-Vincentelli, Chair

Today systems industries face significant challenges to bring products to market. Companies struggle to integrate into complex designs a large number of subsystems designed by various manufacturers. The number of recent high-profile recalls afflicting the automotive and avionics industries is a testament to the difficulty of system design.

The theory of contracts has been proposed to address these issues. Assume-guarantee (AG) contracts are specifications for components in a system that state what is expected from the environment in which the component operates and what is required from the object under specification provided that the environment meets the assumptions of the contract. A contract can thus be expressed as a pair $\mathcal{C} = (A, G)$ of formal properties: the assumptions A made on the environment and the guarantees G required from the component. By asking a third-party to implement a formal specification given as a contract, a system integrator knows *a priori* that the resulting system will meet its specifications. Thus, contracts can serve as the technical counterpart of the legal bindings between parties in a supply chain.

As AG contracts are specifications for components, the theory must offer means of manipulating specifications corresponding to tasks carried out in system design. There were three operations known on contracts before the work presented in this thesis: composition, conjunction, and disjunction. Composition is the algebraic operation which enables compositional design. It gives us the specification of a system built by components that obey the specifications being composed. Conjunction allows for concurrent design of the same object. When different groups are in charge of designing or characterizing different aspects, or viewpoints, of the same design element, they will produce specifications corresponding to the various viewpoints of that object. The operation of conjunction allows us to generate one specification that summarizes these diverse specifications.

Over the course of the research that led to this thesis, we introduced the operation of merging, or strong merging, which gives the specification of an object consisting of multiple

viewpoints when each viewpoint's guarantees is required to hold simultaneously. In addition, we obtained adjoint operations for all previously mentioned operations. For example, the adjoint of composition, called quotient, yields the most relaxed specification which can be composed with the specification of a partial implementation of the design in order to meet a top-level spec. As a result of the work presented here, today we know eight operations on AG contracts and their closed-form expressions.

AG contracts require that the assumptions and guarantees of the pair (A, G) be specified as what in formal methods are called *trace properties*. This type of properties can express many requirements, such as safety (i.e., bad things don't happen) and liveness (i.e., good things eventually happen), but there are many statements we would like to make over systems that these properties cannot express. For instance, it has been shown that several security attributes, such as non-interference and observational determinism, cannot be expressed as trace properties. To express them, we need hyperproperties.

In this context, we introduce in this thesis the concept of hypercontracts. Hypercontracts are a fully-developed AG theory which allows us to reason compositionally about systems using pairs (A, G) of arbitrary hyperproperties. This theory extends the reach of AG reasoning to all system attributes we know how to formalize. This includes all known properties used in security and in machine learning.

Contents

List of Figures

List of Tables

List of Algorithms

Acknowledgments

I am grateful to my advisor, Alberto Sangiovanni-Vincentelli, and to the members of my committee, Albert Benveniste, Francesco Borrelli, and Sanjit A. Seshia, for their guidance and support. I thank Alberto for his trust in welcoming me to his team. I am influenced by his research vision for system design. I benefited from several conversations with Albert; he suggested many research directions. Sanjit pointed out key research questions addressed in this thesis. Edward A. Lee chaired the committee of my quals exam, for which I am grateful.

My time at Berkeley has been filled with much joy. Shirley Salanio, Susanne Kauer, and Jessica Gamble were always very kind to me and answered my administrative questions thoroughly. I am privileged to have met my fellow graduate students and postdocs.

I thank the institutions that funded the completion of this work: UC Berkeley, NSF, DARPA, and Toyota. I thank the University of Verona and the Embassy of France in the United States for grants that enabled me to spend time working in Verona and Rennes, respectively.

Chapter 1

Introduction

In the late 1950s, A. W. Wymore was carrying out research in numerical analysis at the University of Arizona. He recalls a visit from his dean and hearing from him:

> I have just returned from an exciting meeting of the American Society for Engineering Education where I heard a paper on the new discipline of systems engineering. It is no longer sufficient for engineers merely to design boxes such as computers with the expectation that they would become components of larger, more complex systems... We must learn how to design large-scale, complex systems from the top down so that the specification for each component is derivable from the requirements for the overall system. We must also take a much larger view of systems. We must design the man-machine interfaces and even the system-society interfaces [205].

Wymore was then asked to create the first academic department in systems engineering, which began operations in 1960. Trained in mathematics, he sought first to define formalisms to reason about systems [205]. For this he drew inspiration from *Automata Studies*, edited by Shannon and McCarthy [186]. After modeling, he faced the issue of unspecified requirements in system design. This came up vividly during a visit to Lockheed-Georgia as a consultant in 1969. Thus, Wymore came across two key issues of system design.

A third fundamental challenge is system integration. After all, systems are, by definition, comprised of parts, or as Sage and Lynch put it, "it is the integration of subsystems and components that give systems their superiority over a set of elements that do not work together without integration" [173].

We now discuss these three aspects of the field, namely, modeling, specification, and integration.

1.1 Challenges of system design

1.1.1 Modeling

Mathematical models are used to virtually analyze and compose a system before it is actualized in the physical world. By not requiring physical changes to the design in order to yield insight, models enable faster virtual design iterations than could be achieved otherwise. Models enable engineering organizations to analyze a design and understand tradeoffs among competing requirements before the designs are implemented.

In all cases, the purpose of a model is to simulate what would happen to the object adhering to such model when in performs in a setting corresponding to the type of analysis being effected. The quality of the analyses that can be carried out with engineering models depends on the models's faithfulness to reality. If a model matches reality poorly, the analysis effected using such model may not be representative of how the implementation would behave in the real world.

Modeling thus sits at the kernel of system design. One fundamental challenge of system modeling has been the addition of software to systems whose main purpose was not computational, such as transportation media and energy extraction and distribution systems. System capabilities have been greatly increased through the addition of software, but so has their complexity and their potential for displaying undesired behaviors[1]. Designs involving both computational and physical processes whose behavior is determined by both cyber and physical components are called *cyber-physical systems*[2] [130]. Conceptually, the most important contribution of the intellectual enterprise of cyber-physical systems has been to bring to the forefront the challenge of analyzing the interconnection of components modeled using discrete transitions with those using differential equations. It is the opposition between the discrete and the continuous, which René Thom calls "the fundamental aporia of mathematics."

[1]For several examples of system errors in which software was directly involved, see Leveson [133]. She writes, "many of the system safety techniques that have been developed to aid in building electromechanical systems with minimal risk do not seem to apply when computers are introduced. The major reasons appear to stem from the differences between hardware and software and from the lack of system-level approaches to building software-controlled systems." Henzinger and Sifakis say, "the shortcomings of current design, validation, and maintenance processes make software the most costly and least reliable part of embedded applications... We see the main culprit as the lack of rigorous techniques for embedded systems design. At one extreme, computer science research has largely ignored embedded systems, using abstractions that actually remove physical constraints from consideration. At the other, embedded systems design goes beyond the traditional expertise of electrical engineers because computation and software are integral parts of embedded systems" [85].

[2]For reflections on the modeling challenges posed by cyber-physical systems, we refer the reader to the work of E. Lee [123–127]

1.1.2 Specification

D. Hitchins [86] claims that most system engineering practitioners see system design as a process with the following parts: (i) describe the system at the most abstract level, (ii) decompose the requirements functionally, (iii) map decomposed requirements to architecture, and (iv) develop physical elements and integrate. Three of these steps directly address requirements or specifications.

Specifications are syntactic expressions that the design process should map to an engineering deliverable. This deliverable can be an object actualized in the physical world, a software routine, or a mathematical model. Practitioners consistently rank the generation of specifications for a project among the top challenges in system design [151, 173]. The definition of requirements can be an arduous process involving multiple stakeholders; as customers learn new information and react to market conditions, requirements may also change[3].

Requirements can be formal or informal. Specifications in industry are normally written in long documents expressed in natural languages. Writing conflicting or incomplete requirements is not uncommon. Ideally, teams working on different components on a system should be able to interconnect their components if they develop their implementations to meet the component specification. However, if the interfaces are not sufficiently constrained, there may be enough design freedom such that a correct implementation of the component specifications does not allow the implementations to interact with one another. In order to avoid ambiguity, academia argues that formal specifications should replace natural language requirements. Work is needed to bridge formal requirements and statements that would be natural to a designer, so that they are usable.

Formal specifications have well-defined mathematical meaning. They are used in three tasks: verification, synthesis, and testing. In verification, we are interested in proving whether an engineering model satisfies a specification. In synthesis, we pursue algorithms that generate engineering models which are guaranteed to meet the specification. In testing, we subject an engineering model or the implementation to stimuli for which the specification stipulates how the object under test should behave; if the unit under test does not behave according to the specification, we have a violation. These three tasks have probabilistic variants. One can speak of a component having a certain property with some probability or failing a test with some probability.

The task of formally verifying software was first enunciated by Turing in 1949[4]. The program-verification agenda gained force and focus some twenty years later with Floyd [68] and Hoare [87]. An important part of the verification effort is writing the specifications that programs should satisfy. A methodology for this was already reported by Parnas in 1972

[3]For an overview of challenges in requirement engineering, see [80].

[4]His three-page document starts with a familiar paradigm: "How can one check a routine in the sense of making sure that it is right? In order that the man who checks may not have too difficult a task the programmer should make a number of definite assertions which can be checked individually, and from which the correctness of the whole programme easily follows" [195].

[159]. Languages for writing formal specifications were developed[5]. Concurrency introduced a new angle to writing specifications: it is not sufficient anymore to specify an input/output relation for a concurrently-executing routine, as this maintains an ongoing relation with its environment. This realization led to the definition of a reactive system by Harel and Pnueli in 1985 [81]. In the late 90s, it was further recognized that a similar formalism (the behavioral approach) can be used to specify other aspects of complex engineering systems. W. Damm [48] argued that in an engineering design we can consider specifications as "rich components," i.e., collections of specifications of multiple aspects of a design element, like functionality, timing, and power.

1.1.3 Integration

Integration has seen steep increases in magnitude in the 20th century, bringing technical challenges to engineering (how do we design, build, and maintain such systems?)[6] and organizational challenges to the business landscape (how does the organization change to support the development, construction, and maintenance of such systems?)[7]. The organiza-

[5]For example, Pnueli's LTL [166], Clarke and Emerson's CTL [44], Koymans's MTL [109], and Maler and Nickovic's STL [142]

[6]We read, for example: "We are increasingly experiencing a new type of accident that arises in the interactions among components (electromechanical, digital, and human) rather than in the failure of individual components" [132]. "Almost all SI [system integration] failures occur at interfaces primarily due to incomplete, inconsistent, or misunderstood specifications" [141]. "System specification and integration is particularly critical for Original Equipment Manufacturers (OEM) managing the integration and maintenance process with subsystems that come from different suppliers who use different design methods, different software architectures, and different hardware platforms... even inside an OEM itself, complex systems involve a number of different aspects or viewpoints that are generally handled by different teams using different paradigms and tools" [19]. "It is no longer feasible for an individual to fully comprehend and tightly control all the details of design of a complex, electromechanical system... This recognition has led to two trends with unfortunate side effects: (a) an increase in specialization among system engineers... and (b) an increasing use of integrated product teams (IPTs) to bring these sub-specialties together... The increasing specialization has resulted in smaller engineering units and introduced barriers to communication among and across design organizations. Concomitantly, the complexity of modern designs has produced a loss of emphasis on core engineering fundamentals. This effect is evident in the poor translation of user requirements into achievable system specifications; inadequate design for manufacturing and test; and fundamental weakness in addressing design factors that drive system reliability and availability" [152].

[7]Hobday et al. [89] argue that "systems integration has evolved beyond its original technical and operational tasks to encompass a strategic business dimension becoming, therefore, a core capability of many high-technology corporations... The more complex, high-technology, and high cost the product, the more significant systems integration becomes to the productive activity of the firm... Systems integration capabilities are inextricably linked to decisions on whether to make in-house, outsource, or collaborate in production and competition." The authors conclude that "in high-volume products... firms use their capabilities to achieve competitive advantage by exploiting upstream component supplier relations in ways which differ according to the particular phase of each product life cycle. By contrast, in low-volume, high-cost capital goods, manufacturing firms are focusing more on exploiting downstream relationships with system users by integrating services such as maintenance, finance, consultancy, and operations within their product offerings. In both cases, systems integration capability enables firms to move selectively, and simultaneously, up- and

tional challenges of systems engineering are so salient that some authors categorize the field as a branch of management [65, 97].

Some challenges with integration intersect the other topics of systems engineering: How can we use the properties of components to understand the properties of the system? Under what conditions can we obtain emergent phenomena, i.e., system behaviors that could not be anticipated from the components? How can we specify components to make sure that when they are implemented, they will integrate correctly into the system? How do we manage a supply chain with multiple components modeled using various tools/formats and coming from multiple manufacturers?

An important aspect of integration is reuse. Many systems are made of components with similar characteristics. The task of characterizing components so that they can be incorporated in new designs is enticing—but easier said than done [24]. A success story in this venue is digital IC design, in which a library of characterized gates is built and reused across several IC projects.

1.2 Formalizing system design

In order to address the challenges discussed, the literature has proposed a combination of methodologies and mathematical formalisms[8]. Methodologies help to structure the design process, from both a technical and organizational standpoint[9]. They enable a separation of concerns, so engineers can focus their energies on specific aspects of the design, without being distracted by others. Formalisms provide mathematical grounding to design; they go hand in hand with methodologies. Our focus from now on will be on formalisms.

Mathematics mainly takes place on top of two ground theories: set theory and category theory. Georg Cantor introduced the first in the 19th century [31], and Samuel Eilenberg and Saunders Mac Lane introduced the second in the 1940s [64]. Cantor was motivated

downstream to gain advantages in the marketplace." Davies et al. [52] argue that "the ability to integrate a range of components from a variety of internal and external suppliers is becoming the core activity required to provide integrated solutions. The traditional advantages of the vertically-integrated systems seller offering single-vendor designed systems is no longer a major source of competitive advantage in many industries."

[8] A. Sangiovanni-Vincentelli suggests a program [174]: "to deal with system-level problems... the issue to address is... understanding of the principles of system design, the necessary change to design methodologies, and the dynamics of the supply chain." He also calls for more rigor in the design process: "I share... the strong belief that a new design science must be developed to address the challenges listed above where the physical is married to the abstract, where the world of analog signals is coupled with the one of digital processors, and where ubiquitous sensing and actuation make our entire environment safer and more responsive to our needs. SLD [system-level design] should be based on the new design science to address our needs in a fundamental way." J. Sifakis characterized design science "as a formal process encompassing the three stages of requirements expression, proceduralization and materialization" [188]

[9] For example, component-based design [4, 129, 190], platform-based design [57, 67, 105, 174–176], rigorous design [22, 187, 189], platform-based engineering [139, 158], elegant design [76, 140, 200], model-based design [66, 98, 171, 204], set-based design [185, 191], interface-based design [5, 172], and contract-based design [18, 19, 156, 177].

to develop set theory based on his studies of infinity, while Eilenberg and Mac by Lane by problems in algebraic topology[10] [137].

The main compositional modeling formalism used in engineering and computer science we can call *the behavioral model*. In it, we understand components by the behaviors they can exhibit. A component is a set of behaviors. The composition of components is given by set intersection. In the late 1960s, Wymore developed a specification mechanism for systems engineering in which one provides input and output trajectories and an eligibility function that relates inputs to outputs [205]—this is akin to behavioral modeling. In the late 70s, Jan Willems [201, 202] from the controls community pointed out that system modeling should not a priory impose notions of inputs and outputs on signals, and proposed using a sets of behaviors to represent and manipulate components in control systems. In computer science, referring to a paper by Dijkstra from 1965 [59], Lamport writes, "Dijkstra was aware from the beginning of how subtle concurrent algorithms are and how easy it is to get them wrong. He wrote a careful proof of his algorithm. The computational model implicit in his reasoning is that an execution is represented as a sequence of states . . . I have found this to be the most generally useful model of computation—for example, it underlies a Turing machine. I like to call it the standard model" [116]. In the 70s, both A. Pnueli [166] and R. Keller [104] represented state transition systems using the standard model, and C. A. R. Hoare [88] defined processes as the set of traces they generate, showing that various set operations on processes yield other processes. E. A. Lee and A. Sangiovanni-Vincentelli [128] introduced the Tagged Signal Model (TSM) in the second half of the 90s. The TSM defines behaviors with special structure. Using it, they were able to provide formal definitions for slippery concepts, such as synchronicity, which are often hard to discuss due to lack of agreement about their meaning.

The behavioral model is set theoretic. It is the denotation of formal specifications in computer science. There are recent efforts to embed behavioral modeling within frameworks based on category theory [2, 3, 12, 108, 111, 131, 192, 207]. Other authors have provided algebraic formalisms to handle the system design process [9, 23, 35, 150].

1.3 This thesis

This thesis is about the theory of contracts, a formalism that addresses the specification and integration challenges we discussed. This first chapter discussed fundamental challenges in system design. Chapter 2 introduces contracts and discusses the development of the theory and some applications outside the work described in this thesis. The description of historical developments partially comes from

[162]. Passerone, R., Incer, I., and Sangiovanni-Vincentelli, A. L.

[10]For Eilenberg and Mac Lane, the category was not the main concept of their theory. They say, "it should be observed first that the whole concept of a category is essentially an auxiliary one; our basic concepts are essentially those of a functor and of a natural transformation" [64].

Contract model operators for composition and merging: extensions and proofs. Technical Report DISI-19-004, Dipartimento di Ingegneria e Scienza dell'Informazione, University of Trento, August 2019

In Chapter 3, we begin to discuss the contributions that form part of this thesis. The focus of the chapter is a binary operation called quotient. Assume-guarantee contracts have a notion of composition. This operation produces the specification of a system formed by components obeying the contracts being composed. Quotient is the adjoint of composition. Given a top-level specification and the specification of a partial implementation of the design, the quotient gives the most relaxed specification whose composition with the existing partial specification refines the top-level contract. It was not known whether the quotient operation exists for contracts, and here we answer the question affirmatively and provide its closed-form expression and examples of its use. The contents of this chapter are based on the following document:

[96]. Incer, I., Sangiovanni-Vincentelli, A. L., Lin, C.-W., and Kang, E. Quotient for assume-guarantee contracts. In *16th ACM-IEEE International Conference on Formal Methods and Models for System Design* (October 2018), MEMOCODE'18, pp. 67–77

After discussing the quotient of AG contracts, we present in Chapter 4 a general study of the quotient operation in compositional theories. The chapter begins with the intriguing observation that the quotient formulas in several algebraic theories are very similar despite the fact that the theories may be quite different. In other words, it is common in compositional theories to have inequalities of the form $A \cdot x \leq B$, where $\cdot$ is the notion of composition, and $\leq$ is a notion of refinement. We often want to solve for the largest x that satisfies the inequality; this largest x is called quotient. It turns out in many theories this quotient obeys the syntactic expression $\gamma\left(\gamma(A) \cdot B\right)$, where γ is an involution. We propose an algebraic structure called preheap that is guaranteed to have closed-form expressions for the quotient. Many existing compositional theories are preheaps. This discussion is based on the following paper:

[95]. Incer, I., Mangeruca, L., Villa, T., and Sangiovanni-Vincentelli, A. L. The quotient in preorder theories. In *Proceedings 11th International Symposium on Games, Automata, Logics, and Formal Verification, Brussels, Belgium, September 21-22, 2020* (Brussels, Belgium, 2020), J.-F. Raskin and D. Bresolin, Eds., vol. 326 of *Electronic Proceedings in Theoretical Computer Science*, Open Publishing Association, pp. 216–233

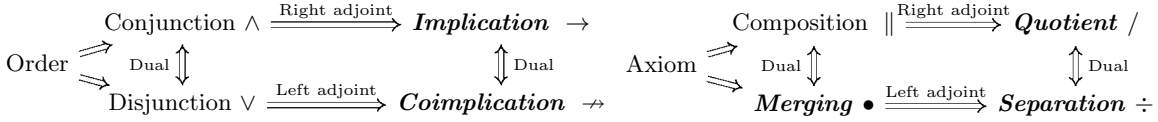
One of the central aspects of assume-guarantee contracts is their ability to represent and handle several viewpoints, or aspects, of a design. To the same component we can assign several contracts, one for each viewpoint. The publication that introduced assume-guarantee contracts [18] argued that we can summarize multiple contract viewpoints into a

single contract through the operation of conjunction. Chapter 5 discusses another binary operation, called merging, which sometimes produces a merger of viewpoints in a way that is more intuitive for designers. We also discuss the adjoint operation to merging, called separation, and present examples of the use of both operations. The chapter is based on

> [161]. Passerone, R., Incer, I., and Sangiovanni-Vincentelli, A. L. Coherent extension, composition, and merging operators in contract models for system design. *ACM Trans. Embed. Comput. Syst. 18*, 5s (Oct. 2019)

Chapter 6 summarizes the algebraic aspects of assume-guarantee contracts. It can be used as a self-contained reference to all known binary operations on contracts. Two operations, implication and coimplication, which are the adjoints of conjunction and disjunction, respectively, have not been published before, as far as I know. This chapter also studies various algebraic structures—monoids and semirings—within contracts, and the maps between these structures. Much of this chapter is new material. After having considered the algebraic aspects of assume-guarantee contracts, Chapter 7 discusses the syntactic representation of contracts and algorithms to manipulate contracts using such representations. The content is new.

As a result of the work presented here, today we know eight operations on AG contracts and their closed-form expressions. The following diagram shows these operations on AG contracts. Two are derived from the partial order of contracts, and two are generated axiomatically. The five operations shown bold were introduced (reintroduced, in the case of merging) in the work covered in this thesis.



We shift gears in Chapter 8. Up to this point, we have treated contracts as represented by a pair of trace properties. While powerful, trace properties are unable to express certain important requirements, such as observational determinism and secure information flow—key properties studied in security. We introduce the concept of hypercontracts to represent environments and guarantees using arbitrary structured hyperproperties. The amount of structure in the hyperproperties is left to the user. The content in this chapter is taken from the following documents:

> [94]. Incer, I., Benveniste, A., Sangiovanni-Vincentelli, A. L., and Seshia, S. A. Hypercontracts. In *NASA Formal Methods* (Cham, 2022), J. Deshmukh, K. Havelund, and I. Perez, Eds., Springer International Publishing

> [93]. Incer, I., Benveniste, A., Sangiovanni-Vincentelli, A. L., and Seshia, S. A. Hypercontracts. *arXiv preprint arXiv:2106.02449* (2021)

There is one appendix containing proofs skipped in the main body of the text.

Chapter 2

Introduction to contracts

This chapter introduces assume-guarantee contracts and contract-based design and discusses the work carried out on the theory outside the work presented in this thesis.

2.1 Historical background

2.1.1 Assume-guarantee reasoning

The notion of contracts derives from the theory of abstract data types and was first suggested by Meyer in the context of the programming language Eiffel [148, 149], following the original ideas introduced by Floyd and Hoare [68, 87] to assign logical meaning to sequential imperative programs in the form of triples of assertions. A Hoare triple $\{A\}c\{B\}$ consists of properties A and B and a program c. The program is correct if B holds after the execution of c, assuming that A holds before this execution. Meyer introduces *preconditions* and *postconditions* as assertions or specifications for the methods of a class, and *invariants* for the class itself. Preconditions correspond to the assumptions under which the method operates, while postconditions express the promises at method termination, provided that the assumptions are satisfied. Promises must be guaranteed only if the assumptions are satisfied. Invariants, on the other hands, are conditions that must be true of the state of the class regardless of any assumption. The notion of class inheritance, in this case, is used as a refinement, or sub-typing, relation. To guarantee safe substitutability, a subclass is only allowed to weaken assumptions and to strengthen promises and invariants. Similar ideas were already present in seminal work by Dijkstra [60] and Lamport [115] on *weakest preconditions* and *predicate transformers* for sequential and concurrent programs, and in more recent work by Back and von Wright, who introduce contracts in the *refinement calculus* [11]. In this formalism, processes are described with guarded commands operating on shared variables. Contracts are composed of *assertions* (higher-order state predicates) and *state transformers*. These contracts are of a very different nature, since there is no clear indication of the role (assumption or promise) a state predicate or a state transformer may play. This formalism

is best suited to reason about discrete, un-timed process behavior.

The work of Dill on asynchronous trace structures was the first to differentiate between acceptable and non-acceptable uses of a component [61]. Behaviors, or traces, can be either accepted as *successes*, or rejected as *failures*. The failures, which are still possible behaviors of the system, correspond to unacceptable inputs from the environment, and are therefore the complement of the assumptions. Safe substitutability is expressed as trace containment between the successes and failures of the specification and the implementation. The conditions obtained by Dill are equivalent to requiring that the implementation weaken the assumptions of the specification while strengthening the promises. Composition is taken as the product with requirement relaxation. The relaxation mechanism is obtained through a process called *autofailure manifestation* and *failure exclusion*, which yield a maximal interface. Wolf later extended the same technique to a discrete synchronous model [203]. The trace structures developed by Dill and Wolf address the problem of receptiveness, or input completeness, by proving closure properties and by giving decision procedures. Finally, Process Spaces [154] is a more general model proposed by Negulescu following the work of Dill and Wolf, and is based on pairs of sets: X is the set of possible behaviors, and Y is the set of acceptable behaviors. A process has the requirement that $X \cup Y = \mathcal{B}$. Process Spaces define the operations of product and exclusive sum, which are syntactically the same as the contract operations of parallel composition and merging, respectively. Product is used to compose design elements, but no insight is given into the use of exclusive sum.

Since the late 70s, L. Lamport worked actively on writing specifications for computer programs. He quickly noticed the importance of separating specifications into what the component is supposed to do and the assumptions that the component makes on its environment in order to operate correctly. He says, "to write a specification, there must be an object to be specified and a well-defined interface between the object and its environment" [114]. The work of M. Abadi and L. Lamport [1] made the fundamental contribution of providing a principle that tells how to compose two specifications of components written in such a way that they split responsibilities between the design component and the environment in which it is instantiated. In this work, the authors focus on the formulation of the specification as an *implication*. In particular, while a specification is allowed to make assumptions, it is *not* interpreted as constraining the environment, or else the specification is considered unrealizable. At the same time, however, and following the work of Dill [61], realizability is defined as a game, a technique that would later be employed in other popular interface models, such as interface automata [54]. Of greater significance, when defining the refinement relation, the authors insist that assumptions be weakened, which is unnecessary for implications, thus effectively obtaining the equivalent of an interface model. The main result of Abadi and Lamport is a full set of proof rules that show when the parallel composition of components satisfies a given property, under a set of assumptions. These proof rules have later been reformulated in similar ways in several other contract models and tools [15, 42, 63, 74]. Composition, expressed as intersection of behaviors, takes primarily the view of the component.

2.1.2 Interface theories

In 1998, de Alfaro and Henzinger introduced interface automata, a "light-weight formalism for capturing temporal aspects of software component interfaces which are beyond the reach of traditional type systems" [54]. The formalism used by the authors has several distinguishing traits: the choice of an automata-based language, the partition of symbols into inputs and outputs, a new refinement relation, the separation of concerns between assumptions from the environment and responsibilities of the object under specification, and an optimistic approach to composing automata. Syntactically, interface automata are indistinguishable from IO automata [136]. They differ in their associated semantic concepts of refinement and parallel composition. In contrast to IO automata, which require their implementations to be receptive to input actions, IA state the assumptions on the environments in which valid implementations run. In other words, certain moves by the environment may not be allowed by the IA. The notion of refinement for interface automata is the alternating simulation of Alur et al. [6]. Compared to the usual definition of simulation for automata, alternating simulation builds on a game view of automata, in which inputs are seen as adversarial. IA are composed according to an optimistic approach: for two IA to be composed, it suffices that there be an environment that allows both to operate.

Interface automata gave rise to a series of *interface theories* which extended IA in multiple ways, for example, modal I/O automata [119], resource interfaces [37], timed interfaces [56], timed IO automata [51], permissive interfaces [84], modal interfaces [170], and interfaces with support for component reuse [62]. Moreover, interface automata influenced approaches to formally design and analyze cyber-physical systems.

The classic Interface Automata [54] and HRC [20, 50] models are similar to synchronous trace structures, where failures are implicitly all the traces that are not accepted. Thus, the interface is maximal. Composition is defined on automata, rather than on traces, and requires a procedure similar to requirement relaxation (and therefore to autofailure manifestation) in order to maintain maximality.

These concepts were subsequently developed to maturity by giving rise to MDE (*Model Driven Engineering*) [103, 122, 180]. In this context, interfaces are described as part of the system architecture and comprise typed ports, parameters and attributes. Contracts on interfaces are typically formulated in terms of constraints on the entities of components, using the Object Constraint Language (OCL) [77, 198]. Roughly speaking, an OCL statement refers to a context for the considered statement, and expresses properties to be satisfied by this context (e.g., if the context is a class, a property might be an attribute). Arithmetic or set-theoretic operations can be used in expressing these properties. To account for behavior and performance, the classical approach in MDE consists in enriching components with *methods* that can be invoked from outside, and/or *state machines*. Attributes on port methods have been used to represent non-functional requirements or provisions of a component [30]. The effect of a method is made precise by the actual code that is executed when calling this method. The state machine description and the methods together provide directly an implementation for the component—several MDE related tools, such as GME and

Rational Rose, automatically generate executable code from this specification. The notion of refinement is replaced by the concept of class inheritance. Inheritance, however, is unable to cover aspects related to behavior refinement, since it is limited to constraining the signature of the method, rather than their behavior. Nor is it made precise what it means to take the conjunction of interfaces, only approximated by multiple inheritance, or to compose them. Liskov and Wing [134] address some of these shortcomings by strengthening Meyer's contract model to include the specification of extra methods in the subtype, which may change the object state, and to account for the preservation of history properties.

2.1.3 Algebraic interface theories

Bauer et al. [15] present a meta-theory for interface-based design. The objective is to provide a method with which to construct a *contract* framework given a *specification* framework with sufficient reasonable properties. The idea is to devise a set of axiomatic definitions for the contract operators and relations, with provable compositional properties, which can be instantiated given a specification theory. The work focuses on the relation of refinement and defines operators for composition, conjunction, and quotient. In particular, they show how to constructively define the composition operator. Their method is based on the use of canonical forms, and treats environments and implementations asymmetrically. The modal specification model introduced by Raclet et al. [170] is used as a case study.

Chilton et al. [40] develop an algebraic theory of interface automata which is also useful to shed light on the properties of the operators and relations. The formalism is reminiscent of Dill's trace structures, and extends that work with additional operators, such as quotient. The authors also address issues of progress in the context of finite traces, unlike trace structures which use infinite traces. Of particular interest is the definition of refinement, which allows the refining component to have signatures with different sets of inputs and outputs. Consequently, conjunction can also be defined on components with different signatures. This facilitates a multiple viewpoint approach.

2.1.4 From software to cyber-physical systems

The control community had developed since the early 1990's a Cyber Physical System (CPS) modeling approach based on ODEs plus discrete-time dynamical systems, exemplified by the Simulink tool. Solution trajectories of such models turn out to be *traces* in the computer science setting. Similarly, the tradition of making statements over the behaviors (i.e. the standard model) became well established in computer science during the 1970s. The trace was the object that enabled us to conclude whether a program satisfied a certain requirement. Traces were given convenient syntactical representations with temporal logics, notably Pnueli's LTL [166] and Clarke and Emerson's CTL [44]. Many algebraic formalisms were also introduced to reason about systems whose components were modeled as collections of traces. Thus, the concept of trace crosses the different communities contributing to the design of CPSs.

One of the contributions of interface automata was to refocus on the interfaces of reactive systems; this way, it brought new vigor to research in compositional design. Damm [48] introduced to systems engineering the notion of a *rich component*: one can carry out model-based design using functional and non-functional aspects of the design. This idea motivated the agenda of applying the techniques of formal methods to any kind of CPS, a theory and methodology called contract-based design [18, 177].

Building on top of trace-based modeling, Assume-Guarantee (AG) contracts [18, 19] provide a formal framework to contract-based design, in which assumptions and guarantees are first-class citizens. AG contracts assume that all functional and non-functional behaviors of the system have been modeled in advance. We assume an underlying set $\mathcal{B}$ of behaviors, generalizing traces or executions dealt with in the computer science literature. The mathematical bases of AG contracts are the same as those for process spaces [153]. While Negulescu thought of process spaces as representing specifications for arbitrary software components, AG contracts were introduced to model arbitrary functional and non-functional aspects of CPSs.

More specifically, contracts are behavioral specifications $\mathcal{C} = (A, G)$ such that $A \cup G = \mathcal{B}$. They are assigned to components in a design such that components are required to meet their guarantees G when the environment in which they are instantiated meets the assumptions A. Contracts support, among others, an operation of composition, which yields the specification of the concurrent operation of two components adhering to two different contracts, and an operation of "viewpoint merging," which yields a single specification when two specifications are provided for the same design element. Contracts also have a refinement relation, which tells when one contract specification is more specific than another. If a component adheres to a strict specification, it adheres to any more relaxed specification. Later, Benveniste et al. introduced a meta-theory of contracts to frame several models in the same formalism [19], and discuss their operators.

Tripakis et al. [194] also study the connection between different kinds of interface specifications. In particular, they show how to transform Relational Interfaces [193], which are not input complete (or receptive), into an equivalent set of input complete specifications, in order to avoid game-theoretic methods and have a more efficient analysis. We believe this procedure is akin to going from Interface Automata to Trace Structures. Similarly, Carmona and Kleijn [32] explore the issue of compatibility in a general multi-component settings. This work deals primarily with questions of receptiveness, progress and deadlock freedom. However, the authors do not develop a full interface or contract model, but express assumptions implicitly in terms of the actions which are enabled at each state of the components.

Damm et al. [49] make a distinction between weak and strong assumptions. Mangeruca et al. [145] use a similar notion, called *precondition*, to define the conditions under which the promises must hold, in a form similar to implications. The authors use this concept to define the *completeness* of a contract relative to the requirements, and avoid implementations that vacuously satisfy their contract. The formalism is also used to define extensions of the contract, by properly combining the promises and their preconditions. The authors also provide an operator to override a promise by another promise.

P. Nuzzo et al. show how to use contracts and currently-available tools in the design of complex systems [155–157]. A. Iannopollo et al. [91, 92] describe a software package capable of finding refinements for a contract expressed in LTL by composing objects from a library of LTL contracts.

2.2 Problems addressed by contracts

Having considered the historical development of contracts, we now detail the issues addressed by assume-guarantee contracts for system analysis and design. Chapter 1 discussed major challenges in system design: modeling, specification, and integration. Assume-guarantee contracts are formal specifications attached to components in a system; these specifications have structure: they state (i) what is assumed from the environment and (ii) what is required from the component when the environment meets the contract's specifications. Contracts directly address system specification and integration. Assume-guarantee contracts are orthogonal to modeling. Contracts begin to exist after modeling decisions about components have been effected.

Contracts can be embedded within the design methodology of platform-based design [174]. In platform-based design, a top-level specification is mapped to an implementation in a sequence of refinement steps, each adding increasing amounts of detail to the implementation. At each step, a specification is mapped to a composition of elements from a library. If the level of abstraction at that step (or platform) permits the designer to capture all necessary details, the design process ends at that platform; otherwise, the output from that implementation step becomes the specification for the next one. In platform-based design, it is possible for one platform to lack enough elements to implement a given specification. When that happens, we identify a missing component that needs to be added to the library so that the specification is implemented.

In contract-based design [177], the specifications just mentioned are expressed as contracts, as are the elements from the library of components. Thus, we can implement a system by focusing on the specifications of the devices comprising the system. This yields two questions: when a family of contracts from the library is chosen as the "implementation" of the top-level step, what do we mean formally? When a family of contracts is selected to implement a top-level spec, what is the contract of the system they generate?

The answer to the first question requires the theory to have a way of comparing contracts; this will tell us when a contract specification is more specific, or more stringent, than another. This notion will be called refinement. The second question forces the theory of contracts to provide a binary operation on contracts which yields the system obtained by the simultaneous operation of objects adhering to the contracts being composed. This last sentence also means that we must have ways of relating components to contracts, so that we can check whether they are valid implementations of the specification.

In the sense we discussed, we keep specifications separate because they correspond to different components. A different reason for keeping contracts separate is the handling of

viewpoints. It can be the case the several groups work on different aspects of the same design element. For example, one group may be in charge of developing the functionality specification of the object, while another develops the specification corresponding to the performance characterization of the object. We end up with two specifications corresponding to the same design element. Contracts have an operation, called conjunction, which summarizes into a single contract all the specifications of the same design element.

We now proceed to discuss the theory of contracts. As contracts as formal specifications, we first describe the behavioral modeling formalism.

2.3 Behavioral modeling

Behavioral modeling understands components as the set of behaviors they can display. What is a behavior? We can say that it is a formalized execution of a component, an instance of the component operating. Any type of component used in engineering design supports a mathematical description. To illustrate the notion of a behavior, consider the voltage amplifier shown on the left in Figure 2.1. The device has input x and output y. The amount of detail that should be present in modeling is always just that which enables us to answer the questions we have—not more. Suppose we are only interested in the static operation of the amplifier. In that case, the notion of time is not needed in our description. If we had a perfect amplifier whose output is exactly equal to its input, we could say that the behaviors are all pairs of real numbers (x, y) such that $y = x$. In this case, each of these pairs is a behavior. Formally, we would express the amplifier as

$$M = \left\{ (x, y) \in \mathbb{R}^2 \;\middle|\; y = x \right\}.$$

If we were interested in expressing dynamic attributes of a component, i.e., how it changes over time, the behaviors we just used would be insufficient. Now we would need to tell how the inputs and outputs, and possibly state variables, vary over time. If we assume that time is a continuous variable that takes values in the nonnegative real numbers, $\mathbb{R}^{\geq 0}$, then each behavior of the amplifier could be expressed as a function with domain $\mathbb{R}^{\geq 0}$ and codomain $\mathbb{R}^2$ (for the variables x and y). We could write the amplifier as

$$M = \left\{ f \in \mathbb{R}^{\geq 0} \to \mathbb{R}^2 \;\middle|\; f_1(t) = f_2(t) \text{ for all } t \in \mathbb{R}^{\geq 0} \right\},$$

where the subscript notation means the projection of the codomain $\mathbb{R}^2$ to the corresponding component (we assume the first component is x, and the second y).

The examples we considered are continuous, but we can also behaviorally model discrete systems, such as M'', shown on the right in Figure 2.1. Here the indicated variables could be thought of as floating point numbers, and the operations as arithmetic over these numbers; the flops capture the state of the variables at certain times. M'' can be written as

$$M'' = \left\{ f \in \mathbb{N} \to F^4 \;\middle|\; \text{For all } t \in \mathbb{N}, \begin{array}{l} f_3(t+1) = f_1(t) + f_2(t) \text{ and} \\ f_4(i+2) = f_3(i+1) f_2(i) \end{array} \right\},$$

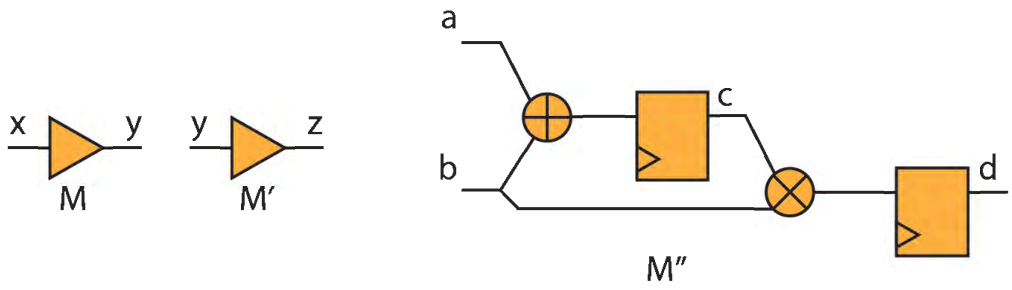


Figure 2.1: Examples of engineering systems to be modeled behaviorally

where F is the set of floating-point numbers.

Observe that the functional notation for the representation of behaviors allows us to abstract whether the behaviors are continuous or discrete, whether their domain is bounded or not. Having available the notion of a component, we discuss the notion of a *property*. Properties, like components, are usually defined as sets of behaviors. The difference is semantics: a property tells the kinds of behaviors which, according to a criterion, are deemed acceptable. For example, we may place into a property all behaviors deemed safe or responsive. A component M is said to satisfy a property P if all its behaviors are members of said property, i.e., if M is a subset of P. Given a property P and a model M, the task of determining whether $M \subseteq P$ is called *model-checking*, an important and extensive subject which is not the scope of the present chapter. We suggest [43] as a reference.

The purpose of modeling components is to obtain insight from them or from the systems we can build out of them. Model checking would yield information about components. In order to reason about systems out of components, we need two notions: a notion to relate components to each other and another to represent systems obtained from the composition of multiple components. We will now increase the formality of our exposition to set the theory of contracts on a firm basis, and the definitions above will give us the language needed to discuss these concepts. First, contracts will provide logical machinery to reason about specifications; thus, we assume that the modeling tasks have been completed, and we have decided on the types of behaviors over which we want to make predicates. This brings us to our first definition:

Definition 2.3.1. *Let $\mathcal{B}$ be a set whose elements we call behaviors.*

By defining this set, we assume that we have already decided on the formalism we use to model our components and the types of properties we wish to verify for those components. The notions of a component, and of a property, follow immediately:

Definition 2.3.2. *A component is a subset of $\mathcal{B}$. Similarly, a property is a subset of $\mathcal{B}$.*

The difference between a component and a property is how we use them: we think of components as sets containing the behaviors that a design entity can display, and we

understand a property as the collection of behaviors having a quality of interest, like safety or liveness. When we speak of a component having a property, we mean this:

Definition 2.3.3. *Suppose M is a component, and P a property. We say M satisfies P, written $M \models P$, when*

$$M \subseteq P.$$

In other words, M satisfies the property when all behaviors of M have a quality of interest. This notion allows us to define an order for components:

Definition 2.3.4. *Let M and M' be components. We say that M is a subcomponent of M', written $M \leq M'$, when M satisfies all properties of M', i.e., when*

$$M \subseteq M'.$$

We understand a subcomponent as one meeting more stringent requirements. Recall our component M from Figure 2.1. A refinement of this contract would be, for example, the component

$$\left\{ f \in \mathbb{R}^{\geq 0} \to \mathbb{R}^2 \;\middle|\; f_1(t) = f_2(t) \text{ and } f_1(t) \leq 5 \text{ for all } t \in \mathbb{R}^{\geq 0} \right\}.$$

One may verify that the behaviors of this component belong to those of the amplifier.

Now that we have the notion of refinement, we consider the construction of systems. We model systems as components interacting with one another. The notion of composition in behavioral modeling is as follows.

Definition 2.3.5. *Let M and M' be components. The composite of M and M', denoted $M \parallel M'$, is the component*

$$M \parallel M' = M \cap M'.$$

If we interpret M as the behaviors satisfying a certain constraint, and M' as those satisfying another, the composite is the component containing the behaviors satisfying both constraints simultaneously. This notion of composition is independent of the topology of the connection between M and M'; the topology is implicitly included in the collections of behaviors of each component.

Consider components M and M' from Figure 2.1. When we discuss the composition of components, it is necessary that all their behaviors be defined with respect to the same variables, that is, both M and M' must refer to variables x, y, z. Thus, we could express components M and M' as follows:

$$M = \left\{ f \in \mathbb{R}^{\geq 0} \to \mathbb{R}^3 \;\middle|\; f_1(t) = f_2(t) \text{ for all } t \in \mathbb{R}^{\geq 0} \right\}$$
$$M' = \left\{ f \in \mathbb{R}^{\geq 0} \to \mathbb{R}^3 \;\middle|\; f_2(t) = f_3(t) \text{ for all } t \in \mathbb{R}^{\geq 0} \right\}.$$

Observe that M imposes no restrictions on the third component (corresponding to variable z), and M' imposes no restrictions on the first (corresponding to variable x). The composition of these two objects is

$$M \parallel M' = \left\{ f \in \mathbb{R}^{\geq 0} \to \mathbb{R}^3 \;\middle|\; f_1(t) = f_2(t) \text{ and } f_2(t) = f_3(t) \text{ for all } t \in \mathbb{R}^{\geq 0} \right\},$$

which corresponds to our understanding that $z = y = x$ when the two components are used concurrently. Observe that the variables used by the components determine the connection between them, i.e., due to the fact the output of M is y and the input of M' is y, the composition of M and M' connects the output of M to the input of M'.

Now we consider the notion of component replacement in a system.

Proposition 2.3.6. *Let M be a component and M_s be a subcomponent of M. For any component M',*

$$M' \parallel M_s \leq M' \parallel M.$$

This proposition is a direct consequence of the monotonicity of set intersection with respect to the subset order and has important implications. It says that if we build a system as the composite $M' \parallel M$, in this system we can replace M with its subcomponent M_s, and the new system will satisfy all properties it satisfied before the component replacement happened.

2.4 Assume-guarantee contracts

Assume-guarantee contracts are a pairs of properties $\mathcal{C} = (A, G)$, where $A, G \subseteq \mathcal{B}$, the universe of behaviors. We call A the assumptions of the contract, and G the guarantees of the contract. We say that a component E is an environment for a contract $\mathcal{C} = (A, G)$ if E satisfies the assumptions of the contract (i.e., $E \models A$). We say that a component M is an implementation for a contract $\mathcal{C}$ if M satisfies the guarantees of the contract, provided that it operates in an environment of the contract (i.e., $M \parallel E \models G$ for all $E \models A$).

We split formal specifications into assumptions and guarantees implementations can only make promises when some conditions hold on the environments in which they operate. By splitting a specification into assumptions and guarantees, we partially characterize the context of operations of implementations, and we state that implementations should fulfill their promises at least in those contexts.

Contracts provide formal structure to support two key system tasks: compositional design and indepedent analysis. Compositional design has to do with building a whole using constituents. Independent analysis means that contracts allows engineers to use for a given kind of analysis only the amount of detail necessary to carry out that task.

2.4.1 Compositional design

As we pointed out at the beginning of the chapter, we need two concepts to implement compositional design with contracts:

- a way to compare contracts and
- an operation to obtain system specifications from the specifications of subsystems.

The notion of order for contracts is called *refinement*.

Definition 2.4.1. *Given contracts $\mathcal{C} = (A, G)$ and $\mathcal{C}' = (A', G')$, we say that $\mathcal{C}$ refines $\mathcal{C}'$, written $\mathcal{C} \leq \mathcal{C}'$, if the environments of $\mathcal{C}'$ are environments of $\mathcal{C}$ and the implementations of $\mathcal{C}$ are implementations of $\mathcal{C}'$.*

In terms of the components of the contracts, we have $\mathcal{C} \leq \mathcal{C}'$ when

$$A' \leq A \quad \text{and} \quad G \cup \neg A \leq G' \cup \neg A'.$$

The notion of refinement allows to say that two contracts are equivalent when they have the same environments and the same implementations. We observe that a contract $\mathcal{C} = (A, G)$ is equivalent to the contract $(A, G \cup \neg A)$. When the pair $(A, G) = (A, G \cup \neg A)$, we say that the contract is in saturated, or canonical, form. Algebraically, contract operations are cleanest when we assume that the contracts are given in canonical form. Thus, from now on, we assume that all contracts are in this form, unless we say otherwise.

When we have a top-level specification $\mathcal{C}$ we wish to implement, and we have implemented a system with specification $\mathcal{C}_s$. The question of whether our system spec meets the top-level is whether $\mathcal{C}_s \leq \mathcal{C}$.

The second notion we needed to implement a compositional design methodology is an operation of composition. The definition of composition follows the Abadi-Lamport composition axiom [1].

Definition 2.4.2. *Given contracts $\mathcal{C}$ and $\mathcal{C}'$, their composition, written $\mathcal{C} \parallel \mathcal{C}'$, is the smallest contract in the refinement order such that*

- *any composition of implementations of the contracts being composed is an implementation of the composite specification;*
- *any composition of an implementation of $\mathcal{C}$ with an environment of the system-level contract is an environment for $\mathcal{C}'$;*
- *the same for $\mathcal{C}$.*

The closed form expression for this operation is

$$\mathcal{C} \parallel \mathcal{C}' = (A \cap A' \cup \neg(G \cap G'), G \cap G')\,.$$

Composition is commutative and associative. The operation of composition together with the refinement relation are the formal keys to compositional, independent design. For example, suppose an OEM wishes to implement a system with top-level specification $\mathcal{C}$.

The OEM identifies a sequence of subsystems obeying specifications $\{\mathcal{C}_i\}_{i=1}^N$ such that the composition of these specifications refines the top-level contract:

$$\mathcal{C}_1 \parallel \ldots \parallel \mathcal{C}_N \leq \mathcal{C}.$$

Then the OEM can hand out the subsystem contracts to independent suppliers. The suppliers can generate implementations for each subcontract. Contract theory guarantees that the system obtained when composing the various implementations will implement the original top-level spec, $\mathcal{C}$. Thus, contracts can formalize interactions between various players in a supply chain.

2.4.2 Independent analysis

The third concept we discussed in the introduction was handling multiple viewpoints. In general, we can associate several specifications to a design element, representing various aspects of the same object. For example, for the same element, we can have a specification $\mathcal{C}^f$ for functionality and $\mathcal{C}^t$ for timing. The overall specification for the object is one which enforces both specifications simultaneously, namely the conjunction in the refinement order:

$$\mathcal{C}^f \wedge \mathcal{C}^t.$$

This conjunction operation is the least-upper bound of the refinement order. If we write $\mathcal{C} = (A, G)$ and $\mathcal{C}' = (A', G')$, the closed form for this operation is

$$\mathcal{C} \wedge \mathcal{C}' = (A \cup A', G \cap G').$$

Now suppose we build a system using two components with specifications $\mathcal{C}_1$ and $\mathcal{C}_2$. Each of these specifications consists of a functionality and a timing spec, i.e., $\mathcal{C}_i = \mathcal{C}_i^f \wedge \mathcal{C}_i^t$. The resulting system is

$$\mathcal{C}_1 \parallel \mathcal{C}_2 = (\mathcal{C}_1^f \wedge \mathcal{C}_1^t) \parallel (\mathcal{C}_2^f \wedge \mathcal{C}_2^t).$$

Property 6 in Ch 4 of [19] tells us that the previous expression can be bounded as

$$\mathcal{C}_1 \parallel \mathcal{C}_2 \leq (\mathcal{C}_1^f \parallel \mathcal{C}_2^f) \wedge (\mathcal{C}_1^t \parallel \mathcal{C}_2^t).$$

In other words, using contracts, we can carry out analysis using only the viewpoints that we are interested in analyzing. The resulting contracts obtained in this way are a correct abstraction of the system specification.

2.4.3 Product lines

The operation of conjunction was derived from the partial order of contracts. The partial order generates a second binary operation, disjunction (or the least upper bound), computed as follows:

$$\mathcal{C} \vee \mathcal{C}' = (A \cap A', G \cup G').$$

This operation plays a role in product lines. Suppose we have several products belonging to the same family of products. Each of the products has its own contract. Then the disjunction of these contracts yields the specification of the product family[1].

[1]We thank Jean-Marc Jézéquel for pointing out during a talk at Inria Rennes that disjunction finds application in product families.

Chapter 3

Quotient for assume-guarantee Contracts

This chapter discusses the notion of quotient set for a pair of contracts and the operation of quotient for assume-guarantee contracts. The quotient set and its related operation can be used in any compositional methodology where design requirements are mapped into a set of components in a library. In particular, they can be used for the so called *missing component* problem, where the given components are not capable of discharging the obligations of the requirements. In this case, the quotient operation identifies the contract for a component that, if added to the original set, makes the resulting system fulfill the requirements.

3.1 Introduction

Decomposing specifications is a recurrent step in reuse-based, meet-in-the-middle design methodologies such as Platform-Based Design (e.g., see [13]). In these methodologies that are fairly common in industry, during the top-down phase, the specification for a system is decomposed into a set of refined specifications of sub-components, i.e., the high-level architecture of the design is determined. This step fits in a refinement-driven process where higher level specifications are mapped into lower level implementations. This decomposition is "guided" by the existence of a set of predefined components in a library, the bottom-up part of the methodology.

More formally, suppose a designer wishes to implement a system that satisfies a top-level specification $\mathcal{T}$, and will use in this design a set of n components from a library with specifications $\mathcal{F} = (\mathcal{T}_i)_{i=1}^n$. If the composition of these n design elements refines the top level specification $\mathcal{T}$, the design assembled from the components satisfies the specification. However, if this is not the case, the designer must add at least one element to the library. In other words, the designer must identify a specification $\mathcal{T}_M$ such that its composition with the composition of $(\mathcal{T}_i)_{i=1}^n$ refines $\mathcal{T}$.

This problem corresponds to identifying the *missing (unknown) component* in a library

(e.g., see [197] and references therein). In the language of contract-based design, we need to compute the *contract quotient* that guarantees that $\mathcal{T}_M$ is as "large" a specification as possible in the refinement order so that whoever is in charge of adding the element to the library has the maximum degree of freedom in its implementation. In industry, this problem is in general tackled heuristically. The notion of quotient for assume-guarantee contracts aims at finding a rigorous procedure for the determination of the largest specification of the missing component.

Related work. In [121], Le et al. address the problem of fixing a decomposition so that it refines a specification: given a top-level contract $\mathcal{C}$ and family of contracts $(\mathcal{C}_i)_{i=1}^n$ whose composition may not refine $\mathcal{C}$, find a family $(\mathcal{C}'_i)_{i=1}^n$ such that $\mathcal{C}'_i \leq \mathcal{C}_i$ for all i and $\|_{i=1}^n \mathcal{C}'_i \leq \mathcal{C}$. In this setting, the designer begins with a high-level specification $\mathcal{C}$ and an initial decomposition of the specification into various components, but the specification, $\mathcal{C}_i$, of any component may need to be corrected to $\mathcal{C}'_i$.

Notions similar to quotients have been previously investigated in the context of various behavioral formalisms. Chilton et al. formulate in [41] an assume-guarantee framework for reasoning about components modeled as a variant of interface automata introduced by Chen et al. [38], including quotient operations. Similarly, Bhaduri and Ramesh [21] investigate the problem of synthesizing, given P and Q as interface automata, R such that the composition of R and P refines Q; they provide a game-theoretic formulation of the problem as computing winning strategies over a game between P and Q. In addition, Raclet [168] introduces the concept of a *residual specification* in the context of modal automata.

Another related line of research is on assumption generation in the context of compositional verification [7, 25, 46]: Given a component M and a desired property P, what is the weakest assumption A that M can make about its environment such that $M \oplus A \models P$? These works typically assume M and A to be labelled transition systems, and exploit their structures as part of a learning algorithm (e.g., L* [8]).

Compared to the contributions mentioned above, our approach differs in that we provide a general form for the contract quotient for assume-guarantee contracts, i.e., a formulation that holds for all variants of assume-guarantee contracts. Furthermore, as far as we know, our approach is the first to introduce the notion of a *quotient set* in the context of assume-guarantee contracts. This defines a range of contracts that can be composed with C_1 to yield the largest contract that refines C.

Specific Contributions. Benveniste et al. in their comprehensive review of contract based design [19], on page 188, state that "no Least Upper Bound and no quotient are known for AG contracts." This chapter provides an explicit form for the quotient operation of AG contracts (Theorem 3.2.5 in Section 3.2). We point out that notions similar to quotients have been proposed for AG contracts before, but these operations are defined when contracts are expressed in specific formalisms [41, 46, 71].

We also introduce the notion of the quotient set (Definition 3.2.6), a set that contains all contracts $\mathcal{C}'$ such that $\mathcal{C}' \parallel \mathcal{C}_1 = (\mathcal{C}/\mathcal{C}_1) \parallel \mathcal{C}_1$ for given contracts $\mathcal{C}$ and $\mathcal{C}_1$. The quotient set tells us which contracts *extend* $\mathcal{C}_1$ into $\mathcal{C}$ in the largest way possible. We fully characterize the quotient set in Theorem 3.2.7 (Section 3.2).

A further contribution of this chapter is an alternate definition of the quotient operation for the meta-theory of contracts (Section 3.3). The new definition makes the quotient operation an obvious analog of the composition operation and makes the derivation of the quotient for AG contracts almost immediate.

Finally, we show a methodology and examples of the use of the quotient operation in the design of an ALU and of an automotive system (Section 3.4). We believe the theory in this chapter can significantly improve the integration process in practical system design. For example, it can assist automotive Original Equipment Manufacturers (OEMs) in defining the specification of a component to be implemented[1]. We show that with appropriate automation tools, which are left as future work, the process can be formal but still efficient.

3.2 Quotient of Assume-Guarantee Contracts

Assume that $\mathcal{C} = (A, G)$ is the top-level specification we wish to implement and that $\mathcal{C}_1 = (A_1, G_1)$ is the specification of a component that will be used in the design. We are interested in the part of the specification that is not discharged by $\mathcal{C}_1$, and in the maximal specification we can form by composing $\mathcal{C}_1$ with a missing specification with the result refining $\mathcal{C}$. We introduce the quotient operation of assume-guarantee contracts in Section 3.2.1. We introduce in Section 3.2.2 the *quotient set* (Definition 3.2.6), a notion that expresses what we need to do to complement $\mathcal{C}_1$ to satisfy $\mathcal{C}$ in a maximal way in the refinement order. Theorems 3.2.5 and 3.2.7 are our main results in this section. Theorem 3.2.5 provides an explicit formula for the quotient operation for assume-guarantee contracts, and Theorem 3.2.7 provides a complete characterization of the quotient set as an interval of contracts. Throughout this section, we assume all AG contracts are given in saturated form. We observe that this is not restrictive since a contract can always be saturated through the mapping $(A, G) \longmapsto (A, G \cup \neg A)$. A contract is semantically equivalent to its saturated form. In our manipulations, we evaluate set-theoretic operators in the order $\neg$, $\cap$, $\cup$. Missing proofs are given in the appendix.

3.2.1 Quotient Operation of AG Contracts

In this section we introduce the quotient operation of AG contracts. Given contracts $\mathcal{C}$ and $\mathcal{C}_1$, the quotient $\mathcal{C}/\mathcal{C}_1$ is the largest contract whose composition with $\mathcal{C}_1$ refines $\mathcal{C}$, i.e.,

$$\mathcal{C}' \leq \mathcal{C}/\mathcal{C}_1 \iff \mathcal{C}' \parallel \mathcal{C}_1 \leq \mathcal{C}. \tag{3.1}$$

The explicit formulation of the quotient is not known for AG contracts [19]. One of our key contributions is the explicit expression for this quotient.

[1] In industrial designs, OEMs generally keep most of the components and only change as few components as possible when a new design is developed. The task here is to perform the minimum amount of work needed to identify and implement the new components

To obtain the quotient, we first define contract R. Lemma 3.2.4 shows that R has properties akin to those of the quotient operation, and Theorem 3.2.5 uses this result to provide the quotient operation.

Definition 3.2.1. *Let $\mathcal{C} = (A, G)$ be an AG contract. We use the notation $a(\mathcal{C})$ and $g(\mathcal{C})$ to refer to the sets of assumptions and guarantees of $\mathcal{C}$, respectively, i.e., $(A, G) \overset{a}{\longmapsto} A$ and $(A, G) \overset{g}{\longmapsto} G$.*

We provide various bounds on a saturated contract $\mathcal{C}'$ that satisfies $\mathcal{C}' \parallel C_1 \leq \mathcal{C}$.

Lemma 3.2.2. *Let $\mathcal{C} = (A, G)$, $\mathcal{C}_1 = (A_1, G_1)$, and $\mathcal{C}' = (A', G')$ be saturated AG contracts such that $\mathcal{C}' \parallel \mathcal{C}_1 \leq \mathcal{C}$. Then*

$$g(\mathcal{C}') \subseteq \neg G_1 \cup G, \tag{3.2}$$
$$g(\mathcal{C}' \parallel \mathcal{C}_1) \subseteq G_1 \cap G, \tag{3.3}$$
$$A \cup \neg G_1 \subseteq a(\mathcal{C}' \parallel \mathcal{C}_1), \quad \textit{and} \tag{3.4}$$
$$A \cap G_1 \subseteq a(\mathcal{C}'). \tag{3.5}$$

Proof. Suppose $G' \not\subseteq G \cup \neg G_1$, i.e., $\emptyset \neq G' \cap \neg(G \cup \neg G_1) = G' \cap \neg G \cap G_1$. Then $G' \cap G_1 \not\subseteq G$, which contradicts $\mathcal{C}' \parallel \mathcal{C}_1 \leq \mathcal{C}$. Thus, $G' \subseteq G \cup \neg G_1$, proving (3.2), and $G' \cap G_1 \subseteq (G \cup \neg G_1) \cap G_1 = G \cap G_1$, and (3.3) holds. Now we prove (3.4). Since $\mathcal{C}' \parallel \mathcal{C}_1 \leq \mathcal{C}$, we have

$$A \subseteq (A_1 \cap A') \cup \neg G_1 \cup \neg G' \tag{3.6}$$

For (3.6) to hold, we must have

$$\begin{aligned} \neg G' &\supseteq A \cap \neg\left((A_1 \cap A') \cup \neg G_1\right) \\ &= A \cap G_1 \cap (\neg A_1 \cup \neg A'). \end{aligned}$$

Equation (3.6) becomes

$$\begin{aligned} &(A_1 \cap A') \cup \neg G_1 \cup \neg G' \\ &\quad \supseteq (A_1 \cap A') \cup \neg G_1 \cup (A \cap G_1 \cap \neg(A_1 \cap A')) \\ &\quad = (A_1 \cap A') \cup \neg G_1 \cup (A \cap G_1 \cap \neg(A_1 \cap A')) \\ &\quad \cup (A \cap G_1) \cup A = A \cup \neg G_1 \cup (A_1 \cap A') \supseteq A \cup \neg G_1, \end{aligned}$$

proving (3.4). Now, from (3.6), $A \cap G' \cap G_1 \subseteq A_1 \cap A' \Rightarrow A \cap G' \cap G_1 \subseteq A'$. Due to the saturation of $\mathcal{C}'$, we also have $A' \supseteq \neg G'$; then $A' \supseteq A \cap G' \cap G_1 \cup \neg G' = A \cap G_1 \cup \neg G' \Rightarrow A' \supseteq A \cap G_1$, proving (3.5). □

We proceed to define contract R, which is computed from $\mathcal{C}$ and $\mathcal{C}'$. We show in Lemma 3.2.4 that contract R has properties that resemble those of the quotient operation, and in Theorem 3.2.5 we use this result to provide the quotient operation.

Definition 3.2.3. *Let $\mathcal{C} = (A, G)$ and $\mathcal{C}_1 = (A_1, G_1)$ be saturated contracts. We define the contract*

$$R(\mathcal{C}, \mathcal{C}_1) := (A \cap G_1, G \cup \neg G_1)\,.$$

When the context of $\mathcal{C}$ and $\mathcal{C}_1$ is clear, we may use the notation $R = R(\mathcal{C}, \mathcal{C}_1)$.

Lemma 3.2.4. *Let $\mathcal{C}$, $\mathcal{C}_1$, and $\mathcal{C}'$ be saturated contracts. The following statements are equivalent:*

i. $\mathcal{C}' \parallel \mathcal{C}_1 \leq \mathcal{C}$ and

ii. $\mathcal{C}' \leq R$ and $A \cap G' \subseteq A_1$.

Proof. (i. $\Rightarrow$ ii.). Let $\mathcal{C}'$ be a saturated contract such that $\mathcal{C}' \parallel \mathcal{C}_1 \leq \mathcal{C}$. From Lemma 3.2.2, $g(\mathcal{C}') \subseteq G \cup \neg G_1 = g(R)$ and $a(\mathcal{C}') \supseteq A \cap G_1 = a(R)$, i.e., $\mathcal{C}' \leq R$. Moreover, from i., $A \subseteq A_1 \cap A' \cup \neg G_1 \cup \neg G' \subseteq A_1 \cup \neg G'$. Intersecting both sides with G', we obtain $A \cap G' \subseteq A_1$.

(ii. $\Rightarrow$ i.). Assume $\mathcal{C}' \leq R$ and $A \cap G' \subseteq A_1$. Then $g(\mathcal{C}' \parallel \mathcal{C}_1) = g(\mathcal{C}') \cap G_1 \subseteq g(R) \cap G_1 \subseteq g(\mathcal{C})$. From ii., we have $A \cap G' \subseteq A_1$ and $A \cap G_1 = a(R) \subseteq a(\mathcal{C}') = A'$; thus, $A_1 \cap A' \supseteq A \cap G' \cap G_1$. Therefore, we can write $A_1 \cap A' = A_1 \cap A' \cup A \cap G' \cap G_1$ (since we are just adding a subset). With this identity, we have $a(\mathcal{C}' \parallel \mathcal{C}_1) = A_1 \cap A' \cup \neg G' \cup \neg G_1 = A_1 \cap A' \cup \neg G' \cup \neg G_1 \cup A \cap G' \cap G_1 \supseteq A = a(\mathcal{C})$. We conclude that $\mathcal{C}' \parallel \mathcal{C}_1 \leq \mathcal{C}$. □

Theorem 3.2.5 (Quotient of AG contracts)**.** *Given saturated contracts $\mathcal{C}$ and $\mathcal{C}_1$, the operation defined as*

$$\mathcal{C}/\mathcal{C}_1 := (A \cap G_1, A_1 \cap G \cup \neg(A \cap G_1)) \tag{3.7}$$

is the quotient in the formalism of assume-guarantee contracts.

Proof. We observe that the operation just defined produces a contract in saturated form. We wish to show that this operation satisfies (3.1).

($\Leftarrow$) in (3.1). Suppose $\mathcal{C}'$ is a saturated contract such that $\mathcal{C}_1 \parallel \mathcal{C}' \leq \mathcal{C}$. By Lemma 3.2.4, $A \cap g(\mathcal{C}') \subseteq A_1 \Rightarrow g(\mathcal{C}') \subseteq A_1 \cup \neg A$ and $\mathcal{C}' \leq R \Rightarrow g(\mathcal{C}') \subseteq G \cup \neg G_1$. Thus, $g(\mathcal{C}') \subseteq (A_1 \cup \neg A) \cap (G \cup \neg G_1) = g(\mathcal{C}/\mathcal{C}_1)$. Moreover, since $\mathcal{C}' \leq R$, it follows that $a(\mathcal{C}') \supseteq A \cap G_1 = a(\mathcal{C}/\mathcal{C}_1)$. We thus conclude that $\mathcal{C}' \leq \mathcal{C}/\mathcal{C}_1$.

($\Rightarrow$) in (3.1). Suppose $\mathcal{C}'$ is a saturated contract such that $\mathcal{C}' \leq \mathcal{C}/\mathcal{C}_1$. We have $g\left((\mathcal{C}/\mathcal{C}_1) \parallel \mathcal{C}_1\right) = G_1 \cap A_1 \cap G \cup G_1 \cap \neg A \subseteq g(\mathcal{C})$. We also have $a\left((\mathcal{C}/\mathcal{C}_1) \parallel \mathcal{C}_1\right) = A_1 \cap A \cap G_1 \cup \neg G_1 \cup A \cap G_1 \cap (\neg A_1 \cup \neg G) \supseteq A = a(\mathcal{C})$. We conclude that $\mathcal{C} \geq (\mathcal{C}/\mathcal{C}_1) \parallel \mathcal{C}_1 \geq \mathcal{C}' \parallel \mathcal{C}_1$. □

We can develop intuition about the expression we derived. A component which is designed to the specifications of the quotient can use the assumptions of $\mathcal{C}$ and can assume that $\mathcal{C}_1$ will meet its guarantees; thus the assumptions of the quotient are $A \cap G_1$. On the side of guarantees, the component built to the quotient specifications must provide the guarantees of $\mathcal{C}$ and the assumptions of $\mathcal{C}_1$ (to make sure $\mathcal{C}_1$ can meet its guarantees); however, the component can relax its guarantees by the assumptions of $\mathcal{C}$ and by the guarantees of $\mathcal{C}_1$. This results in the guarantees we derived for the quotient.

3.2.2 Quotient Set of AG Contracts

We introduce the concept of quotient set to answer two questions:

- When designing a system to meet a specification $\mathcal{C}$, what is the *biggest* specification achievable if a component with specification $\mathcal{C}_1$ must be used in the design? That is, we seek to characterize the contracts whose compositions with $\mathcal{C}_1$ result in the biggest possible contract that refines $\mathcal{C}$.
- Given a contract $\mathcal{C}_1$ and a contract $\mathcal{C}$ that is the result of the composition of $\mathcal{C}_1$ with another contract, what are all contracts $\mathcal{C}'$ that satisfy $\mathcal{C} = \mathcal{C}_1 \parallel \mathcal{C}'$? In this case, we solve the inverse problem to composition.

We call the set of such contracts the *quotient set*. We begin our quest for the quotient set with an observation: suppose that $\mathcal{C}'$ is a saturated contract that satisfies $\mathcal{C}' \parallel \mathcal{C}_1 \leq \mathcal{C}$. Then $\mathcal{C}' \leq \mathcal{C}/\mathcal{C}_1$. This last statement implies that $\mathcal{C}' \parallel \mathcal{C}_1 \leq \mathcal{C}/\mathcal{C}_1 \parallel \mathcal{C}_1$. Thus, the greatest resulting contract achievable by composing $\mathcal{C}_1$ with a contract with the result refining $\mathcal{C}$ is $\mathcal{C}_1 \parallel \mathcal{C}/\mathcal{C}_1$. A quick computation reveals that

$$\mathcal{C}_1 \parallel \mathcal{C}/\mathcal{C}_1 = (A \cup \neg G_1, G_1 \cap (A_1 \cap G \cup \neg A)) .$$

Thus, we define a set Q whose elements are saturated contracts whose composition with $\mathcal{C}_1$ is equal to the composition we just derived (Definition 3.2.6). We then show in Theorem 3.2.7 that the quotient set is completely characterized as an interval of contracts. After providing this result, we discuss a special and important case of contract decomposition which allows us to simplify many expressions.

Definition 3.2.6. *Let $\mathcal{C} = (A, G)$ and $\mathcal{C}_1 = (A_1, G_1)$ be saturated contracts. We define the quotient set $Q(\mathcal{C}, \mathcal{C}_1)$ as*

$$\begin{aligned} Q(\mathcal{C}, \mathcal{C}_1) := \{\mathcal{C}' \,|\, &\mathcal{C}' \text{ is saturated}, \\ &g(\mathcal{C}' \parallel \mathcal{C}_1) = G_1 \cap (A_1 \cap G \cup \neg A), \\ &a(\mathcal{C}' \parallel \mathcal{C}_1) = A \cup \neg G_1\}. \end{aligned}$$

We also define the lower quotient operation $(\mathcal{C}/\mathcal{C}_1)_-$ as

$$(\mathcal{C}/\mathcal{C}_1)_- := (A \cup \neg G_1 \cup \neg A_1, G_1 \cap (A_1 \cap G \cup \neg A)).$$

Theorem 3.2.7. *Let $\mathcal{C} = (A, G)$ and $\mathcal{C}_1 = (A_1, G_1)$ be saturated contracts. Then $Q(\mathcal{C}, \mathcal{C}_1) = [(\mathcal{C}/\mathcal{C}_1)_-, \mathcal{C}/\mathcal{C}_1]$.*

Proof. ($\Leftarrow$) Let $\mathcal{C}'$ be a saturated contract with $(\mathcal{C}/\mathcal{C}_1)_- \leq \mathcal{C}' \leq \mathcal{C}/\mathcal{C}_1$. We wish to show that $\mathcal{C}' \in Q$. A quick calculation shows that $\mathcal{C}/\mathcal{C}_1, (\mathcal{C}/\mathcal{C}_1)_- \in Q$, which means that $\mathcal{C}_1 \parallel (\mathcal{C}/\mathcal{C}_1)_- = \mathcal{C}_1 \parallel \mathcal{C}/\mathcal{C}_1 = \mathcal{C}_1 \parallel \mathcal{C}''$ for any $\mathcal{C}'' \in Q$. Composing our assumption with $\mathcal{C}_1$ results

in $\mathcal{C}_1 \parallel (\mathcal{C}/\mathcal{C}_1)_- \leq \mathcal{C}_1 \parallel \mathcal{C}' \leq \mathcal{C}_1 \parallel \mathcal{C}/\mathcal{C}_1$; therefore, $\mathcal{C}_1 \parallel \mathcal{C}' = \mathcal{C}_1 \parallel \mathcal{C}''$, which implies that $\mathcal{C}' \in Q$.

($\Rightarrow$) Let $\mathcal{C}' = (A', G') \in Q$. From the definition of Q, it follows immediately that $\mathcal{C}_1 \parallel \mathcal{C}' \leq \mathcal{C}$. Thus, $\mathcal{C}' \leq \mathcal{C}/\mathcal{C}_1$. We need to show that $(\mathcal{C}/\mathcal{C}_1)_- \leq \mathcal{C}'$. Expanding $\mathcal{C}_1 \parallel \mathcal{C}'$ and using the fact that $\mathcal{C}' \in Q$ gives

$$A \cup \neg G_1 = a(\mathcal{C}_1 \parallel \mathcal{C}') = A_1 \cap A' \cup \neg G_1 \cup \neg G' \text{ and} \tag{3.8}$$

$$G_1 \cap (A_1 \cap G \cup \neg A) = g(\mathcal{C}_1 \parallel \mathcal{C}') = G_1 \cap G'. \tag{3.9}$$

Equation (3.9) gives the following bounds:

$$G_1 \cap (A_1 \cap G \cup \neg A) \subseteq G' \subseteq A_1 \cap G \cup \neg A \cup \neg G_1.$$

Plugging *any* of these bounds in (3.8) results in $A \cup \neg G_1 = A_1 \cap A' \cup \neg G_1 \cup \neg G \cup A \cap \neg A_1$. From this expression, we get the bound $A' \subseteq A \cup \neg G_1 \cup \neg A_1 = a\left((\mathcal{C}/\mathcal{C}_1)_-\right)$. Moreover, we note that the leftmost expression of (3.9) is $g\left((\mathcal{C}/\mathcal{C}_1)_-\right)$; thus, (3.9) gives us $g\left((\mathcal{C}/\mathcal{C}_1)_-\right) \subseteq G'$. We conclude that $(\mathcal{C}/\mathcal{C}_1)_- \leq \mathcal{C}'$. □

It should be emphasized that Theorem 3.2.7 gives a full characterization of the quotient set, which tells the contracts extending $\mathcal{C}_1$ into $\mathcal{C}$ in the largest way possible. The bounds we obtained are the quotient operation and the lower quotient. We understand what the quotient operation gives us (the part of the top-level spec $\mathcal{C}$ that $\mathcal{C}_1$ is missing). But what is the lower quotient? We start by recalling that Q contains the contracts whose composition with $\mathcal{C}_1$ gives the biggest possible contract that refines $\mathcal{C}$; the result of this composition is $\mathcal{C}_1 \parallel \mathcal{C}/\mathcal{C}_1$ since the quotient gives the largest extension of $\mathcal{C}_1$ into $\mathcal{C}$. We showed that $(\mathcal{C}/\mathcal{C}_1)_- \in Q$, so $\mathcal{C}_1 \parallel (\mathcal{C}/\mathcal{C}_1)_-$ is the biggest contract achievable by using $\mathcal{C}_1$ while refining $\mathcal{C}$. We observe that the assumptions of the lower quotient are $A \cup \neg G_1 \cup \neg A_1$, i.e., the lower quotient must fulfill its guarantees when the assumptions of $\mathcal{C}$ are met, when the assumptions of $\mathcal{C}_1$ are not met, or when the guarantees of $\mathcal{C}_1$ are not met; this means that if $\mathcal{C}_1$ fails to behave as it promised, $(\mathcal{C}/\mathcal{C}_1)_-$'s guarantees are in force. And what are these guarantees? If A holds, these guarantees are $G_1 \cap A_1 \cap G$, i.e., the contract meets the guarantees of $\mathcal{C}$ and meets the assumptions and guarantees of $\mathcal{C}_1$; if A does not hold, these guarantees are G_1. Thus, we interpret $(\mathcal{C}/\mathcal{C}_1)_-$ as the contract whose implementations are maximally redundant with respect to contract $\mathcal{C}_1$ while completing $\mathcal{C}_1$ in the largest possible way. In contrast, $\mathcal{C}/\mathcal{C}_1$ relies completely on the fact that $\mathcal{C}_1$ will behave as it promises.

3.2.2.1 A simplification of the quotient

Now we consider a simplification that occurs to the quotient operation when $A \cap G \subseteq A_1$. This condition holds when, for instance, all inputs of $\mathcal{C}_1$ can be mapped directly to either compatible inputs or compatible outputs of $\mathcal{C}$; note that in this case we are interpreting the assumptions and guarantees of $\mathcal{C}_1$ in terms of IO behavior. The following corollary shows how the quotient set and its bound simplify when $A \cap G \subseteq G_1$:

Corollary 3.2.8. *Let $\mathcal{C} = (A, G)$ and $\mathcal{C}_1 = (A_1, G_1)$ be saturated contracts. If $A \cap G \subseteq A_1$, the quotient set simplifies to*

$$\begin{aligned} Q(\mathcal{C}, \mathcal{C}_1) := \{\mathcal{C}' \,|\, &\mathcal{C}' \text{ is saturated}, \\ &g(\mathcal{C}' \parallel \mathcal{C}_1) = G_1 \cap G, \\ &a(\mathcal{C}' \parallel \mathcal{C}_1) = A \cup \neg G_1\}, \end{aligned}$$

the quotient operation becomes $\mathcal{C}/\mathcal{C}_1 = R(\mathcal{C}, \mathcal{C}_1)$, and the lower quotient becomes $(\mathcal{C}/\mathcal{C}_1)_- = L(\mathcal{C}, \mathcal{C}_1)$, where

$$L(\mathcal{C}, \mathcal{C}_1) := (A \cup \neg(G_1 \cap A_1), G \cap G_1)\,.$$

Proof. Let $A \cap G \subset A_1$. Then $G_1 \cap (A_1 \cap G \cup \neg A) = G_1 \cap (A_1 \cap G \cup \neg A \cup G \cap \neg A_1) = G_1 \cap G$.

Therefore, we have

$$\begin{aligned} Q(\mathcal{C}, \mathcal{C}_1) &= \left\{ \mathcal{C}' \,\middle|\, \begin{array}{l} \mathcal{C}' \text{ is saturated}, \\ g(\mathcal{C}' \parallel \mathcal{C}_1) = G_1 \cap (A_1 \cap G \cup \neg A), \\ a(\mathcal{C}' \parallel \mathcal{C}_1) = A \cup \neg G_1 \end{array} \right\} \\ &= \left\{ \mathcal{C}' \,\middle|\, \begin{array}{l} \mathcal{C}' \text{ is saturated}, \\ g(\mathcal{C}' \parallel \mathcal{C}_1) = G_1 \cap G, \\ a(\mathcal{C}' \parallel \mathcal{C}_1) = A \cup \neg G_1 \end{array} \right\}. \end{aligned}$$

Moreover, $(\mathcal{C}/\mathcal{C}_1)_- = (A \cup \neg G_1 \cup \neg A_1, G_1 \cap (A_1 \cap G \cup \neg A)) = (A \cup \neg G_1 \cup \neg A_1, G_1 \cap G) = L$.

Finally, we observe that $A_1 \cap G \cup \neg(A \cap G_1) = A_1 \cap G \cup \neg A \cup \neg G_1 = A_1 \cap G \cup \neg A \cup \neg G_1 \cup G \cap \neg A_1 = G \cup \neg G_1$. We thus have $\mathcal{C}/\mathcal{C}_1 = (A \cap G_1, A_1 \cap G \cup \neg(A \cap G_1)) = (A \cap G_1, G \cup \neg G_1) = R(\mathcal{C}, \mathcal{C}_1)$. □

Corollary 3.2.8 tells us that R is equal to the quotient operation and L is equal to the lower quotient when the condition $A \cap G \subseteq A_1$ holds. Observe the form of R: $R = (A \cap G_1, G \cup \neg G_1)$. It assumes the assumptions of $\mathcal{C}$ and the guarantees of $\mathcal{C}_1$, and it guarantees $g(\mathcal{C})$ relaxed by whatever $\mathcal{C}_1$ guarantees. This is a very intuitive notion of the quotient. And what is L? Note that $L = (A \cup \neg(G_1 \cap A_1), G \cap G_1)$. This contract is responsible for its guarantees when either the assumptions of $\mathcal{C}$ are in force or when either the assumptions or guarantees of $\mathcal{C}_1$ are not met (i.e., when $\mathcal{C}_1$ fails). And the guarantees of L are $G \cap G_1$, i.e., L provides the guarantees of both $\mathcal{C}_1$ and $\mathcal{C}$. Thus, L is correlated with a maximally redundant design.

We now consider a simple example that demonstrates the use of these concepts.

3.2.3 An Illustrative Example

Suppose we are designing a system with a Boolean input r and Boolean outputs s and p. Upon the assertion of r, the purpose of this system is to eventually assert s and to

eventually assert p as long as the environment respects reasonable physical constraints. That is, the system must guarantee $\mathbf{G}(r \longrightarrow \mathbf{F}s \wedge \mathbf{F}p)$, subject to the assumptions $e \in E$ (i.e., a continuous environment variable is within some acceptable set). The top level contract is

$$\mathcal{C} = (A, G) = (e \in E, \mathbf{G}(r \longrightarrow \mathbf{F}s \wedge \mathbf{F}p) \vee e \notin E)\,.$$

Suppose a component to be used in the design guarantees the assertion of s one time event after the assertion of r. Moreover, suppose that this component has laxer requirements on the environment than the top-level spec. It follows this component obeys the contract

$$\mathcal{C}_1 = (A_1, G_1) = (e \in E_1, \mathbf{G}(r \longrightarrow \mathbf{X}s) \vee e \notin E_1)\,,$$

where $E_1 \supseteq E$.

If a component with contract $\mathcal{C}_1$ is used to build a design that meets the spec $\mathcal{C}$, we compute the quotient to determine how much of the top-level spec $\mathcal{C}_1$ is missing. Intuitively, what do we expect the quotient to give us? We see that $\mathcal{C}_1$ can satisfy the $\mathbf{F}s$ part of $\mathcal{C}$. Thus, we expect the quotient to only have to guarantee $\mathbf{G}(r \longrightarrow \mathbf{F}p)$. We now carry out the computation.

Since $A \subseteq A_1$, it follows that R is the contract quotient $\mathcal{C}/\mathcal{C}_1$ (Corollary 3.2.8). We compute $R = (A \cap G_1, G \cup \neg G_1)$:

$$\begin{aligned} a(R) =& e \in E \wedge \mathbf{G}(r \longrightarrow \mathbf{X}s) \vee e \in E \wedge e \notin E_1 \\ g(R) =& \mathbf{G}(r \longrightarrow \mathbf{F}s \wedge \mathbf{F}p) \\ & \vee e \notin E \vee \neg\mathbf{G}(r \longrightarrow \mathbf{X}s) \wedge e \in E_1. \end{aligned}$$

Since $E \subseteq E_1$, we can simplify the assumptions to $a(R) = e \in E \wedge \mathbf{G}(r \longrightarrow \mathbf{X}s)$. The guarantees become

$$\begin{aligned} g(R) &= \mathbf{G}(r \longrightarrow \mathbf{F}s \wedge \mathbf{F}p) \vee e \notin E \vee \neg\mathbf{G}(r \longrightarrow \mathbf{X}s) \\ &= \mathbf{G}\left(r \longrightarrow \mathbf{F}s \wedge \mathbf{F}p\right) \wedge \mathbf{G}\left(r \longrightarrow \mathbf{X}s\right) \\ &\quad \vee e \notin E \vee \neg\mathbf{G}(r \longrightarrow \mathbf{X}s) \\ &= \mathbf{G}\left((r \longrightarrow \mathbf{F}s \wedge \mathbf{F}p) \wedge (r \longrightarrow \mathbf{X}s)\right) \\ &\quad \vee e \notin E \vee \neg\mathbf{G}(r \longrightarrow \mathbf{X}s) \\ &= \mathbf{G}(r \longrightarrow \mathbf{X}s \wedge \mathbf{F}p) \\ &\quad \vee e \notin E \vee \neg\mathbf{G}(r \longrightarrow \mathbf{X}s) \\ &= \mathbf{G}(r \longrightarrow \mathbf{F}p) \vee e \notin E \vee \neg\mathbf{G}(r \longrightarrow \mathbf{X}s). \end{aligned}$$

R is (up to saturation) what we posited was missing: $\mathbf{G}(r \longrightarrow \mathbf{F}p)$.

Computing $L = (A \cup \neg(A_1 \cap G_1), G \cap G_1)$ results in $a(L) = e \in E \vee e \notin E_1 \vee \neg\mathbf{G}(r \longrightarrow \mathbf{X}s)$ and $g(L) = \mathbf{G}(r \longrightarrow \mathbf{X}s \wedge \mathbf{F}p) \vee \mathbf{G}(r \longrightarrow \mathbf{X}s) \wedge e \notin E \vee e \notin E_1$. As we discussed before, L has the characteristic of providing the guarantees of $\mathcal{C}$ and of $\mathcal{C}_1$, i.e., L provides a specification with redundancy.

3.3 Quotient in the Meta-Theory of Contracts

In this section, we consider contracts at their most general level, i.e., we stop considering the assume-guarantee framework and operate on our contracts as defined in the meta-theory of contracts. Our contributions are a new definition for the quotient operation and the introduction of the quotient set in the meta-theory. Our purpose in introducing a new (but equivalent) definition of the quotient operation is that the new formulation is an obvious adjoint of the composition operation, making the definition of quotient symmetrical to the definition of composition. From our definition, the derivation of the quotient operation in the assume-guarantee framework comes readily. Hence, at the end of the section we provide a second derivation of the AG quotient.

3.3.1 What is the Meta-Theory of Contracts?

Before discussing various contract theories, Benveniste et al. [19] introduce a meta-theory of contracts. This meta-theory defines contracts on a set of primitives and allows for a birds-eye view of the subject, focusing on semantic concepts. Several key facts can be proved at this level. We can interpret other contract theories, like assume-guarantee and interface theories, as specializations of the meta-theory.

In the meta-theory, the most primitive concept is the component. Composition, $\times$, is a partial binary operation on components. We say components M_1 and M_2 are *composable* if $M_1 \times M_2$ is well-defined. We say a component E is an environment for component M if $M \times E$ is well-defined. A contract $\mathcal{C}$ has semantics given by $(\mathcal{E}, \mathcal{M})$, where $\mathcal{E}$ and $\mathcal{M}$ are sets of components which are valid environments and implementations, respectively, of the contract $\mathcal{C}$.

A contract $\mathcal{C}$ is called *consistent* if $\mathcal{M} \neq \emptyset$ and *compatible* if $\mathcal{E} \neq \emptyset$. We say that a component M is an implementation of contract $\mathcal{C}$ ($M \models^M \mathcal{C}$) iff $M \in \mathcal{M}$; we say that a component E is an environment of contract $\mathcal{C}$ ($E \models^E \mathcal{C}$) iff $E \in \mathcal{E}$.

Refinement in the meta-theory is defined as follows: we say contract $\mathcal{C}' = (\mathcal{E}', \mathcal{M}')$ is a refinement of contract $\mathcal{C} = (\mathcal{E}, \mathcal{M})$ if $\mathcal{E}' \supseteq \mathcal{E}$ and $\mathcal{M}' \subseteq \mathcal{M}$. For contracts $\mathcal{C}$ and $\mathcal{C}'$, $\mathcal{C} \wedge \mathcal{C}'$ and $\mathcal{C} \vee \mathcal{C}'$ are the Greatest Lower Bound (GLB) and Least Upper Bound (LSB), respectively, in the refinement order. In the meta-theory, we make the following assumption:

Assumption 3.3.1. *Both the GLB and LUB are well-defined.*

Now we get to composition. The composition of contracts $\mathcal{C}_1 = (\mathcal{E}_1, \mathcal{M}_1)$ and $\mathcal{C}' = (\mathcal{E}', \mathcal{M}')$ is defined as follows: $\mathcal{C}_1 \parallel \mathcal{C}'$ is well-defined if M_1 and M' are composable for all $M_1 \in \mathcal{M}_1$ and $M' \in \mathcal{M}'$. If it is well-defined, this composition is $\mathcal{C}_1 \parallel \mathcal{C}' = \wedge \mathbf{C}_{\mathcal{C}_1, \mathcal{C}'}$, where

$$\mathbf{C}_{\mathcal{C}_1,\mathcal{C}'} = \left\{ \mathcal{C} \;\middle|\; \begin{array}{l} M_1 \times E \models^E \mathcal{C}', \quad M' \times E \models^E \mathcal{C}_1, \\ \text{and } M' \times M_1 \models^M \mathcal{C} \text{ for all} \\ E, M_1, M' \text{ such that } E \models^E \mathcal{C}, \\ M_1 \models^M \mathcal{C}_1, \text{ and } M' \models^M \mathcal{C}' \end{array} \right\}.$$

The quotient operation is defined as $\mathcal{C}/\mathcal{C}_1 = \vee\{\mathcal{C}' \,|\, \mathcal{C}' \parallel \mathcal{C}_1 \leq \mathcal{C}\}$, i.e., the greatest $\mathcal{C}'$ such that $\mathcal{C}' \parallel \mathcal{C}_1$ refines $\mathcal{C}$. Any contract that refines the quotient also refines $\mathcal{C}$ when composed with $\mathcal{C}_1$ and vice versa, as expressed in (3.1).

3.3.2 The Quotients in the Meta-Theory

We define the following sets of contracts:

Definition 3.3.1. *Given contracts $\mathcal{C}$ and $\mathcal{C}_1$, let*

- $S := \{\mathcal{C}' \mid \mathcal{C}' \parallel \mathcal{C}_1 \leq \mathcal{C}\}$. *Contracts whose composition with $\mathcal{C}_1$ refines $\mathcal{C}$.*
- $U := \{\mathcal{C}' \parallel \mathcal{C}_1 \mid \mathcal{C}' \in S\}$. *The compositions of $\mathcal{C}_1$ that refine $\mathcal{C}$.*
- $Q := \{\mathcal{C}' \mid \mathcal{C}' \parallel \mathcal{C}_1 = \vee U\}$. *Quotient set: contracts whose composition with $\mathcal{C}_1$ is largest while refining $\mathcal{C}$.*

The definition of quotient in the meta-theory is given by $\mathcal{C}/\mathcal{C}_1 = \vee S$. From the definitions we introduced, it is clear that $\mathcal{C}/\mathcal{C}_1 = \vee Q$. We now introduce another set of contracts which we will show is equivalent to S (Lemma 3.3.3).

Definition 3.3.2. *Given contracts $\mathcal{C}$ and $\mathcal{C}_1$, we define the following set of contracts:*

$$\mathbf{C}_{\mathcal{C}/\mathcal{C}_1} := \left\{ \mathcal{C}' \;\middle|\; \begin{array}{l} M_1 \times E \models^E \mathcal{C}', \quad M' \times E \models^E \mathcal{C}_1, \\ \text{and } M' \times M_1 \models^M \mathcal{C} \text{ for all} \\ E, M_1, M' \text{ such that } E \models^E \mathcal{C}, \\ M_1 \models^M \mathcal{C}_1, \text{ and } M' \models^M \mathcal{C}' \end{array} \right\}.$$

Lemma 3.3.3. $\mathbf{C}_{\mathcal{C}/\mathcal{C}_1} = S$.

Proof. We have $S = \{\mathcal{C}' \mid \mathcal{C}' \parallel \mathcal{C}_1 \leq \mathcal{C}\}$. We observe we can write S as

$$S = \left\{ \mathcal{C}' \;\middle|\; \begin{array}{l} \mathcal{C}' \parallel \mathcal{C}_1 \leq \mathcal{C}, \\ M_1 \times E \models^E \mathcal{C}', \quad M' \times E \models^E \mathcal{C}_1, \\ \text{and } M' \times M_1 \models^M \mathcal{C}_1 \parallel \mathcal{C}' \text{ for all} \\ E, M_1, M' \text{ such that } E \models^E \mathcal{C}_1 \parallel \mathcal{C}', \\ M_1 \models^M \mathcal{C}_1, \text{ and } M' \models^M \mathcal{C}' \end{array} \right\}$$

since we have just conjoined a true statement to the conditions that define the set. Now, since $\mathcal{C}_1 \parallel \mathcal{C}' \leq \mathcal{C}$ within the conditions of S, $E \models^E \mathcal{C} \;\Rightarrow\; E \models^E \mathcal{C}_1 \parallel \mathcal{C}'$. Therefore, for any $M_1 \models^M \mathcal{C}_1$, $M' \models^M \mathcal{C}'$, and $E \models^E \mathcal{C}$, we have $M_1 \times E \models^E \mathcal{C}'$, $M' \times E \models^E \mathcal{C}_1$, and

$M' \times M_1 \models^M \mathcal{C}_1 \parallel \mathcal{C}'$. Using again the fact that $\mathcal{C}_1 \parallel \mathcal{C}' \leq \mathcal{C}$ gives $M' \times M_1 \models^M \mathcal{C}$. Therefore, S becomes

$$S = \left\{ \mathcal{C}' \;\middle|\; \begin{array}{l} \mathcal{C}' \parallel \mathcal{C}_1 \leq \mathcal{C}, \\ M_1 \times E \models^E \mathcal{C}', \quad M' \times E \models^E \mathcal{C}_1, \\ \text{and } M' \times M_1 \models^M \mathcal{C} \text{ for all} \\ E, M_1, M' \text{ such that } E \models^E \mathcal{C}, \\ M_1 \models^M \mathcal{C}_1, \text{ and } M' \models^M \mathcal{C}' \end{array} \right\}.$$

We observe that the second condition means that $\mathcal{C} \in \mathbf{C}_{\mathcal{C}_1,\mathcal{C}'}$. This implies that $\mathcal{C}_1 \parallel \mathcal{C}' = \wedge \mathbf{C}_{\mathcal{C}_1,\mathcal{C}'} \leq \mathcal{C}$, which means that the first condition is redundant. We can thus write S as

$$S = \left\{ \mathcal{C}' \;\middle|\; \begin{array}{l} M_1 \times E \models^E \mathcal{C}', \quad M' \times E \models^E \mathcal{C}_1, \\ \text{and } M' \times M_1 \models^M \mathcal{C} \text{ for all} \\ E, M_1, M' \text{ such that } E \models^E \mathcal{C}, \\ M_1 \models^M \mathcal{C}_1, \text{ and } M' \models^M \mathcal{C}' \end{array} \right\},$$

that is, $S = \mathbf{C}_{\mathcal{C}/\mathcal{C}_1}$.

□

It follows that we can define the contract in the meta-theory as $\mathcal{C}/\mathcal{C}_1 = \vee \mathbf{C}_{\mathcal{C}/\mathcal{C}_1}$. This definition is analogous to the definition of composition ($\mathcal{C}_1 \parallel \mathcal{C}' = \wedge \mathbf{C}_{\mathcal{C}_1,\mathcal{C}'}$). Finally, we can define the lower quotient operation for the meta-theory as $(\mathcal{C}/\mathcal{C}_1)_- = \wedge Q$. We now show that this definition of quotient in the meta-theory readily leads to a derivation of the quotient operation for assume-guarantee contracts.

3.3.3 A second derivation of the quotient operation of assume-guarantee contracts

We use the new definition of the contract quotient to derive the quotient operation of AG contracts. The key is expressing $\mathbf{C}_{\mathcal{C}/\mathcal{C}_1}$ in the AG framework. We do this in the following lemma:

Lemma 3.3.4. *Let $\mathcal{C}$, $\mathcal{C}_1$, and $\mathcal{C}'$ be saturated contracts. $\mathbf{C}_{\mathcal{C}/\mathcal{C}_1}$ in the AG framework is given as follows:*

$$\mathbf{C}_{\mathcal{C}/\mathcal{C}_1} = \left\{ (A', G') \;\middle|\; \begin{array}{l} (A', G') \textit{ is saturated}, \\ A \cap G' \subseteq A_1, \\ A \cap G_1 \subseteq A', \textit{ and} \\ G' \cap G_1 \subseteq G \end{array} \right\}. \tag{3.10}$$

Proof. (This proof follows closely that of Lemma 4 in [19].)

($\subseteq$). Let $\mathcal{C}' = (A', G')$ be a saturated contract in $\mathbf{C}_{\mathcal{C}/\mathcal{C}_1}$. Let $E = A$, $M_1 = G_1$, and $M' = G'$. Since $\mathcal{C}' \in \mathbf{C}_{\mathcal{C}/\mathcal{C}_1}$, we have $M_1 \times E \models^E \mathcal{C}'$, $M' \times E \models^E \mathcal{C}_1$, and $M' \times M_1 \models^M \mathcal{C}$. These three expressions are equivalent to $G_1 \cap A \subseteq A'$, $G' \cap A \subseteq A_1$, and $G' \cap G_1 \subseteq G$, respectively. Hence, $\mathcal{C}'$ belongs to the set on the right hand side of 3.10.

($\supseteq$). Let $\mathcal{C}' = (A', G')$ belong to the set on the right hand side of 3.10. Let $E \subseteq A$, $M_1 \subseteq G'$, and $M_1 \subseteq G_1$. We have

$$\begin{aligned}
&E \times M_1 = E \cap M_1 \subseteq A \cap G_1 \subseteq A' \Rightarrow E \times M_1 \models^E \mathcal{C}' \\
&E \times M' = E \cap M' \subseteq A \cap G' \subseteq A_1 \Rightarrow E \times M' \models^E \mathcal{C}_1 \\
&M_1 \times M' = M_1 \cap M' \subseteq G_1 \cap G' \subseteq G \\
&\quad \Rightarrow M_1 \times M' \models^M \mathcal{C}.
\end{aligned}$$

Thus, $\mathcal{C}' \in \mathbf{C}_{\mathcal{C}/\mathcal{C}_1}$, proving the left set inclusion. □

We said we can define the quotient operation as $\mathcal{C}/\mathcal{C}_1 = \vee \mathbf{C}_{\mathcal{C}/\mathcal{C}_1}$. Using Lemma 3.3.4 we obtain another derivation of the AG quotient operation. First we write the quotient in the AG framework:

$$\mathcal{C}/\mathcal{C}_1 = \max \left\{ (A', G') \;\middle|\; \begin{array}{l} (A', G') \text{ is saturated,} \\ A \cap G' \subseteq A_1, \\ A \cap G_1 \subseteq A', \text{ and} \\ G' \cap G_1 \subseteq G \end{array} \right\}.$$

From this expression, we obtain the assumptions and guarantees:

$$\begin{aligned}
a(\mathcal{C}/\mathcal{C}_1) &= A \cap G_1 \quad \text{and} \\
g(\mathcal{C}/\mathcal{C}_1) &= \max \left\{ G' \;\middle|\; \begin{array}{ll} G' \cap A \subseteq A_1 & \text{and} \\ G' \cap G_1 \subseteq G & \end{array} \right\} \\
&= (A_1 \cup \neg A) \cap (G \cup \neg G_1) \\
&= A_1 \cap G \cup \neg(A \cap G_1).
\end{aligned}$$

This derivation of the quotient operation was possible due to the new definition of the contract operation in the meta-theory of contracts.

3.4 Examples

Theorem 3.2.5 can be used as the basis of a decomposition methodology with contracts: suppose we have a top-level contract $\mathcal{C} = (A, G)$, and one component with contract $\mathcal{C}_1 = (A_1, G_1)$ will be used in the design. Then the set of refinements of $\mathcal{C}/\mathcal{C}_1$ is the same as the set of contracts that, when composed with $\mathcal{C}_1$, refine $\mathcal{C}$. Consequently, in order to synthesize a new contract $\mathcal{C}'$ with the property $\mathcal{C}' \parallel \mathcal{C}_1 \leq \mathcal{C}$, we can compute $\mathcal{C}/\mathcal{C}_1$, and we know that

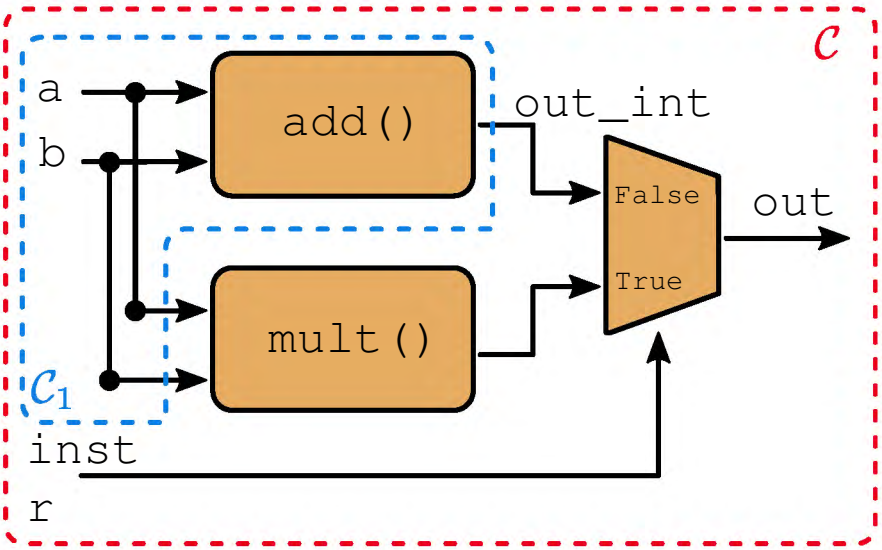


Figure 3.1: ALU diagram of example 3.4.1. The ALU is required to output either the sum or the product of the inputs `a` and `b`, according to the value of the input `inst`: if `inst` is deasserted, the output is the sum; otherwise, it is the product. Suppose there is a component available that implements the addition part of the specification. We expect the quotient to indicate that the multiplication remains to be implemented. We use monospace font to refer to constants, variables, and functions.

any refinement of $\mathcal{C}/\mathcal{C}_1$ will have the property we seek. Moreover, if we wish to synthesize $\mathcal{C}'$, we can seek a refinement of the quotient which is expressible with few parameters in order to make synthesis efficient.

We provide two examples of contract decomposition using the quotient. The first is a logic design example dealing with an Arithmetic Logic Unit (ALU), and the second is an automotive application dealing with Cooperative Adaptive Cruise Control (CACC). Our intention is to show that the operations we derived do indeed provide what one would intuit are the missing specifications in a design, assuming that a given component will be used in the final system. To simplify our presentation, both of our examples meet the requirement that $A \cap G \subseteq A_1$, so according to Corollary 3.2.8, the quotient operation (3.7) becomes equal to $R(\mathcal{C}, \mathcal{C}_1)$ (see Definition 3.2.3); thus, we will repeatedly refer to R as the quotient in these examples. Finally, both examples are given in LTL, but we show the propositions explicitly in order to carry out simplifications.

3.4.1 Implementing an ALU

Consider an ALU with the functionality shown in Figure 3.1. The design receives as input numbers `a` and `b`, a trigger signal `r`, and an instruction `inst`. When `r` is asserted, within n time units the output `out` is equal to either the sum or the multiplication of `a` and `b`, according

to the value of `inst`. For this ALU, we can write the top-level contract $\mathcal{C} = (A, G)$, with

$$G = \bigvee_{i=1}^{n} \Big\{ \big((\texttt{r} \wedge \neg\texttt{inst}) \rightarrow \mathbf{X}^i\texttt{out} = \texttt{add(a,b)}\big) \wedge \big((\texttt{r} \wedge \texttt{inst}) \rightarrow \mathbf{X}^i\texttt{out} = \texttt{mult(a,b)}\big) \Big\} \vee \neg A$$

$$\text{and } A = \texttt{r} \rightarrow \Big(\bigwedge_{i=1}^{n} (\mathbf{X}^i\neg\texttt{r}) \wedge (\mathbf{X}^i\texttt{a} = \texttt{a}) \wedge (\mathbf{X}^i\texttt{b} = \texttt{b}) \wedge (\mathbf{X}^i\texttt{inst} = \texttt{inst}) \Big),$$

i.e., we assume that once `r` asserts, `r` will remain deasserted during the next n time ticks, and all other input signals will not change value during this time; the contract guarantees that at some point during the next n clock ticks, the output will be equal to the addition or multiplication of `a` and `b`, according to the value of `inst`.

Now suppose we have a component which outputs the addition of `a` and `b` at the n-th time tick after the assertion of `r`. If we use this component in our design, we expect only the multiplication remains to be implemented. The component we are using in the design obeys the contract $\mathcal{C}_1 = (A_1, G_1)$ defined as follows:

$$G_1 = (\texttt{r} \rightarrow \mathbf{X}^n\texttt{out\_int} = \texttt{add(a,b)}) \vee \neg A_1 \quad \text{and}$$
$$A_1 = \texttt{r} \rightarrow \Big(\bigwedge_{i=1}^{n} (\mathbf{X}^i\neg\texttt{r}) \wedge (\mathbf{X}^i\texttt{a} = \texttt{a}) \wedge (\mathbf{X}^i\texttt{b} = \texttt{b}) \Big).$$

$\mathcal{C}_1$ guarantees that the output `out_int` will be equal to the addition of the inputs `a` and `b` n ticks after the assertion of `r`. Since $\mathcal{C}_1$ guarantees the addition part of the guarantees of $\mathcal{C}$, we expect the quotient operation to allow us to find a contract that only guarantees the multiplication of the inputs since this is what $\mathcal{C}_1$ is missing from $\mathcal{C}$. Let $\alpha = r \rightarrow \bigwedge_{i=1}^{n} (\mathbf{X}^i\texttt{inst} = \texttt{inst})$. We observe that $A = \alpha \wedge A_1$, so $A \subseteq A_1 \subseteq A_1 \cup \neg G_1$, meaning that $R(\mathcal{C}, \mathcal{C}_1) = (A_2, G_2)$ is the quotient operation (Corollary 3.2.8). Define $\beta = (\texttt{r} \rightarrow \mathbf{X}^n\texttt{out\_int} = \texttt{add(a,b)})$. We compute

$$A_2 = A_1 \wedge \alpha \wedge \beta \quad \text{and}$$
$$G_2 = \bigvee_{i=1}^{n} \Big\{ \big((\texttt{r} \wedge \neg\texttt{inst}) \rightarrow \mathbf{X}^i\texttt{out} = \texttt{add(a,b)}\big) \wedge \big((\texttt{r} \wedge \texttt{inst}) \rightarrow \mathbf{X}^i\texttt{out} = \texttt{mult(a,b)}\big) \Big\} \vee \neg A_2.$$

Now we rewrite G_2 as

$$\begin{aligned} G_2 = \bigvee_{i=1}^{n-1} \Big\{ &\big((\mathtt{r} \wedge \neg \mathtt{inst}) \rightarrow \mathbf{X}^i \mathtt{out} = \mathtt{add(a,b)}\big) \wedge \\ &\big((\mathtt{r} \wedge \mathtt{inst}) \rightarrow \mathbf{X}^i \mathtt{out} = \mathtt{mult(a,b)}\big) \Big\} \vee \\ \Big\{ &\big((\mathtt{r} \wedge \neg \mathtt{inst}) \rightarrow \mathbf{X}^n \mathtt{out} = \mathbf{X}^n \mathtt{out\_int}\big) \wedge \\ &\big((\mathtt{r} \wedge \mathtt{inst}) \rightarrow \mathbf{X}^n \mathtt{out} = \mathtt{mult(a,b)}\big) \Big\} \vee \neg A_2. \end{aligned}$$

Since $A \subseteq A_1 \cup \neg G$, any refinement of R refines $\mathcal{C}$ when composed with $\mathcal{C}_1$. We propose the refinement $\mathcal{C}' = (A', G')$ with

$$\begin{aligned} G' = &\big((\mathtt{r} \wedge \neg \mathtt{inst}) \rightarrow \mathbf{X}^n \mathtt{out} = \mathbf{X}^n \mathtt{out\_int}\big) \wedge \\ &\big((\mathtt{r} \wedge \mathtt{inst}) \rightarrow \mathbf{X}^n \mathtt{out} = \mathtt{mult(a,b)}\big) \vee \neg A' \text{ and} \\ A' = &A_1 \wedge \alpha. \end{aligned}$$

Contract $\mathcal{C}'$ is saturated and satisfies $\mathcal{C}' \leq R$. Since we also have $A \subseteq A_1 \cup \neg G$, Corollary 3.2.8 guarantees that $\mathcal{C}' \parallel \mathcal{C}_1 \leq \mathcal{C}$. We observe that $\mathcal{C}'$ only guarantees the multiplication of `a` and `b`, as we intuited, and that it does not make any assumption about `out_int`; in particular, while R keeps a complete description of the arithmetic relationship between `out_int` and the inputs `a` and `b`, this information is hidden from $\mathcal{C}'$.

3.4.2 Connected Vehicles

Connectivity with other vehicles or with the infrastructure can be used to extend the capabilities of a vehicle by increasing the number, types, and quality of sensors that it can access, or by providing more computational capabilities that the vehicle can use to tackle a task.

We consider the application of connectivity to Cooperative Adaptive Cruise Control (CACC). The scenario is that of Figure 3.2: a vehicle $\mathcal{V}_a$, moving with velocity v_a, attempts to drive on the highway at a certain distance from vehicle $\mathcal{V}_b$. As the top-level spec $\mathcal{C} = (A, G)$, we ask that the distance d_r between the two vehicles be contained in intervals that depend on the speed at which $\mathcal{V}_a$ moves. Thus, we have the guarantees

$$G = \bigwedge_{i \in \mathcal{I}} v_a \in V_i \rightarrow d_r \in [L_i^-, L_i^+] \vee \neg A,$$

where V_i stand for ranges of velocities indexed over a set $\mathcal{I}$; L_i^+ and L_i^- are real numbers which serve as the limits of d_r for various values of the speed v_a.

We observe that vehicle $\mathcal{V}_a$, in order to follow vehicle $\mathcal{V}_b$, must know something about its own state and the state of $\mathcal{V}_b$. Let x_i be the state of vehicle $\mathcal{V}_i$. Assume that $\hat{x}_i$ are the

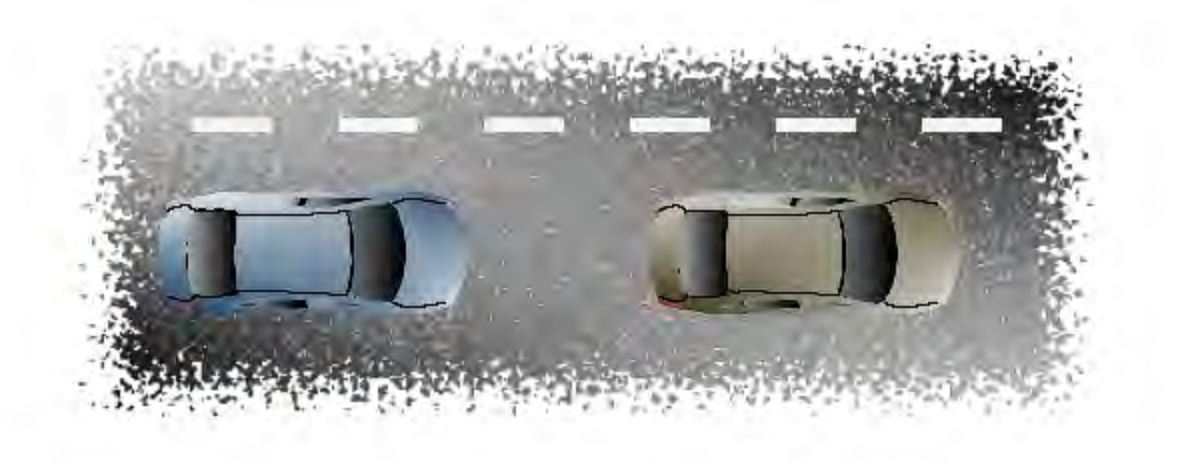


Figure 3.2: In Cooperative Adaptive Cruise Control, the vehicle on the left, $\mathcal{V}_a$, attempts to keep a fixed distance from the other vehicle, $\mathcal{V}_b$. To run its control algorithm, $\mathcal{V}_a$ must use estimates of its own state and of the state of vehicle $\mathcal{V}_b$: the more faithful the estimates, the more reliable the functionality. Suppose that $\mathcal{V}_b$ can estimate its own state much better than $\mathcal{V}_a$ can estimate it. $\mathcal{V}_b$ can share wirelessly the measurements of its state with $\mathcal{V}_a$ for the latter to make better control decisions.

observations which vehicle $\mathcal{V}_a$ makes of the states of each car and which it uses to make control decisions. Note that the x_i are the actual state variables that correspond to reality, but the $\hat{x}_i$ are the state observations that $\mathcal{V}_a$ can access either by making the measurement using its sensors or by receiving data from external sources ($\mathcal{V}_b$ in this case). These observations are affected by the intrinsic accuracy of the sensors, and by the time delays which exist from the analog-to-digital converters that capture physical data to the processors that implement the CACC. These time delays are crucial since, for instance, even if $\mathcal{V}_a$ has access to an extremely accurate velocity measurement of $\mathcal{V}_b$ that was captured 10 minutes ago, the measurement is useless for the purposes of CACC, which operates in real-time.

In order to be able to guarantee its behavior G, $\mathcal{V}_a$ must assume that the states of the vehicles are constrained to reasonable values (e.g., that the speeds of the vehicles are bounded) and that the state observations made by $\mathcal{V}_a$ are faithful to the reality up to a known tolerance. Then, we can write the assumptions for the contract (we assume that operations are evaluated in the order $\neg$, $\wedge$, $\rightarrow$, $\vee$):

$$\begin{aligned} A =& e \in E \wedge x_a \in \mathcal{X}_a \wedge x_b \in \mathcal{X}_b \wedge \\ & \bigwedge_{i \in \mathcal{I}} v_a \in V_i \rightarrow D \in [L_i^-, L_i^+]. \end{aligned}$$

In these expressions, e represents the current state of the environment; E is a set of restrictions of possible behaviors of the environment; the $\mathcal{X}_j$ are restrictions on possible behaviors of the states of the cars; and D is a setting configured by the user and which determines how closely she wishes car $\mathcal{V}_a$ to follow $\mathcal{V}_b$. Note that the contract assumes that the user sets D to a value bounded by the limits L_i^- and L_i^+.

In summary, contract $\mathcal{C}$ assumes that the environment and vehicles behave within certain limits (given by the sets), and guarantees that the relative distance between the cars will be bounded by a certain interval, according to the velocity of the first car. We observe that d_r and v_i can be derived from the state variables x_i.

Suppose that when the driver sets the value of D, the CACC controller of $\mathcal{V}_a$ is capable of making $\mathcal{V}_a$ stay at a distance D from vehicle $\mathcal{V}_b$ up to a tolerance f_ε, i.e., the CACC guarantees $|d_r - D| < f_\varepsilon$. This tolerance f_ε reflects the capability of the CACC control system and depends on the states of both vehicles and on how well the sensors of $\mathcal{V}_a$ match reality. The dependence of f_ε on the states is obvious since it should be harder to keep a fixed distance to a moving target when that target is accelerating, for instance. The dependence of f_ε on the state observations reflects the fact that the more faithfully the estimates match reality, the more reliably the control system behaves. Therefore, f_ε is a function of many arguments: $f_\varepsilon = f_\varepsilon(\|x_a - \hat{x}_a\|, \|x_b - \hat{x}_b\|, x_a, x_b, e)$, where $\|\cdot\|$ is a semi-norm used to tell how well the observations $\hat{x}_i$ match reality x_i. To shorten our expressions, we will simply use f_ε for $f_\varepsilon(\|x_a - \hat{x}_a\|, \|x_b - \hat{x}_b\|, x_a, x_b, e)$. We can write a contract $\mathcal{C}_1 = (A_1, G_1)$ for $\mathcal{V}_a$:

$$\begin{aligned} A_1 =& A, \quad \text{and} \\ G_1 =& |d_r - D| < f_\varepsilon \vee \neg A, \end{aligned}$$

Since $A \subseteq A_1 \cup \neg G$, the refinements of $R = (A_R, G_R)$ also refine $\mathcal{C}$ in composition with $\mathcal{C}_1$. We compute

$$\begin{aligned} A_R =& A \wedge |d_r - D| < f_\varepsilon \quad \text{and} \\ G_R =& \bigwedge_{i \in \mathcal{I}} v_a \in V_i \rightarrow d_r \in [L_i^-, L_i^+] \wedge |d_r - D| < f_\varepsilon \\ & \vee \neg A_R. \end{aligned}$$

Now that we have computed the quotient, we know that R contains the part of $\mathcal{C}$ not met by $\mathcal{C}_1$. Observe that up to this point all manipulations have been mechanical: we have simply computed the quotient starting from the specifications. We are interested, however, in a refinement of the quotient that we can implement, and to find this good refinement we need knowledge from the designer. As designers, we introduce a clause in the guarantees to bound the value of the tolerance. Define the proposition c as follows:

$$c = f_\varepsilon < \min(L_i^+ - D, D - L_i^-).$$

We can now write a contract $\mathcal{C}' = (A', G')$ that refines R using this clause:

$$\begin{aligned} A' =& A_R \quad \text{and} \\ G' =& \bigwedge_{i \in \mathcal{I}} v_a \in V_i \rightarrow d_r \in [L_i^-, L_i^+] \wedge |d_r - D| < f_\varepsilon \wedge c \\ & \vee \neg A', \end{aligned}$$

We observe that $A \subseteq A'$; thus, A' contains the clause $v_a \in V_i \rightarrow D \in [L_i^-, L_i^+]$ for all $i \in \mathcal{I}$. We can use this clause to rewrite G' as follows:

$$G' = \bigwedge_{i \in \mathcal{I}} v_a \in V_i \rightarrow \begin{pmatrix} d_r \in [L_i^-, L_i^+] \wedge \\ |d_r - D| < f_\varepsilon \wedge \\ f_\varepsilon < \min(L_i^+ - D, D - L_i^-) \wedge \\ D \in [L_i^-, L_i^+] \end{pmatrix} \vee \neg A'.$$

The clause $d_r \in [L_i^-, L_i^+]$ is redundant and can thus be eliminated, resulting in

$$G' = \bigwedge_{i \in \mathcal{I}} v_a \in V_i \rightarrow \begin{pmatrix} |d_r - D| < f_\varepsilon \wedge \\ f_\varepsilon < \min(L_i^+ - D, D - L_i^-) \wedge \\ D \in [L_i^-, L_i^+] \end{pmatrix} \vee \neg A'.$$

Finally, since A' contains the clauses $v_a \in V_i \rightarrow D \in [L_i^-, L_i^+]$ and $|d_r - D| < f_\varepsilon$, we can write G' as

$$G' = \bigwedge_{i \in \mathcal{I}} v_a \in V_i \rightarrow f_\varepsilon < \min(L_i^+ - D, D - L_i^-) \vee \neg A'.$$

We observe that the guarantees of G' do not directly refer to d_r; thus, we can further refine $\mathcal{C}'$ with a contract $\mathcal{C}'' = (A'', G'')$ as follows:

$$A'' = A \quad \text{and}$$
$$G'' = \bigwedge_{i \in \mathcal{I}} v_a \in V_i \rightarrow f_\varepsilon < \min(L_i^+ - D, D - L_i^-) \vee \neg A''.$$

Since $\mathcal{C}''$ is a refinement of the quotient, we know it can be used to complete the design. But now we ask, how can we implement contract $\mathcal{C}''$? We started from a high level requirement on the relative distance between the two vehicles and through the quotient obtained a restriction on $f_\varepsilon = f_\varepsilon(\|x_a - \hat{x}_a\|, \|x_b - \hat{x}_b\|, x_a, x_b, e)$, the bounds on $\mathcal{V}_a$'s ability to stay close to $\mathcal{V}_b$. From a design standpoint, a design external to $\mathcal{V}_a$ has no control over the states of the vehicles x_i or over the environment e; however, an external component can impact $\|x_b - \hat{x}_b\|$, i.e., how closely vehicle $\mathcal{V}_a$ can track the state of $\mathcal{V}_b$. While a component of the observation error of x_b by $\mathcal{V}_a$ is comprised of the sensors used, contract $\mathcal{C}''$ can potentially be implemented by guaranteeing that network latency stays within acceptable bounds; that is, the network guarantees that vehicle $\mathcal{V}_a$ always maintains updated information about the state of $\mathcal{V}_b$, enabling the control system of $\mathcal{V}_a$ to make better control decisions.

3.4.3 Observations

We now discuss some commonalities that emerged in the manipulation of contracts in our examples.

1. Once the quotient $\mathcal{C}_q = (A_q, G_q)$ is computed, the next step in a decomposition methodology is to compute a refinement $\mathcal{C}'$ of $\mathcal{C}_q$ (recall that in composition with $\mathcal{C}_1$ (the given component contract), this contract is guaranteed to refine $\mathcal{C}$). To do this, we manipulate the contract $\mathcal{C}_q$ by adding and removing clauses. First, we observe that A_q can often be expressed as a conjunction of clauses, i.e., $A_q = \bigwedge_i a_i$, and in our ALU and automotive examples G_q is of the form $G_q = \bigwedge_j (p_j \rightarrow s_j) \vee \neg A_q$. Observe that $p \rightarrow s \vee \neg a = p \rightarrow s \wedge a \vee \neg a$. This equivalence was used to insert an assumption clause into the guarantees in the examples in order to carry out simplifications in $s \wedge a$. Further, conjoining clauses to s results in a subset of G_q, providing a refinement of $\mathcal{C}_q$.

2. The saturation of the quotient implies that G_q contains all the information needed to carry out simplifications, i.e., no other clauses are needed.

3. Refining the contract quotient involves a notion of what constitutes a good refinement, and set-theoretic operations enriched with simplifications allowed in the formalism in which assumptions and requirements are expressed. The former sets the goal of the simplification process (and thus needs guidance from the designer), while the latter is a systematic mechanism which can be automated. The mechanization of this task is correlated with the identification of the platforms that a given industrial vertical segment will adopt to specify contracts. Here we anticipate that the availability of tools supporting design for a given formalism would stimulate industry adoption. At the same time, a determination of the platforms to be used in an industrial vertical domain would encourage academic research to develop the "right" formalism needed to manipulate contracts for that platform. The question now is how to spark this win-win interaction.

3.5 Summary

We have introduced the notion of quotient set for the theory of contracts, and an explicit formula to compute the quotient operation between two assume-guarantee contracts. We illustrated a decomposition methodology of high-level specifications based on the AG quotient operation. Some lines of future research are the use of the lower quotient operation to devise specifications with redundancy (to ensure reliability) and the creation of software tools that exploit these concepts to decompose specifications.

Chapter 4

Equations over Preorders

The previous chapter discussed the notion of quotient for AG contracts. We now study this notion in a more general setting. Seeking the largest solution to an expression of the form $Ax \leq B$ is a common task in several domains of engineering and computer science. This largest solution is commonly called quotient. Across domains, the meanings of the binary operation and the preorder are quite different, yet the syntax for computing the largest solution is remarkably similar. This chapter is about finding a common framework to reason about quotients. We only assume we operate on a preorder endowed with an abstract monotonic multiplication and an involution. We provide a condition, called admissibility, which guarantees the existence of the quotient, and which yields its closed form. We call preordered heaps those structures satisfying the admissibility condition. We show that many existing theories in computer science are preordered heaps, and we are thus able to derive a quotient for them, subsuming existing solutions when available in the literature. We introduce the concept of sieved heaps to deal with structures which are given over multiple domains of definition. We show that sieved heaps also have well-defined quotients.

4.1 Introduction

The identification of missing objects is a common task in engineering. Suppose an engineer wishes to implement a design with a mathematical description B, and will use a component with a description A to implement this design. In order to find out what needs to be added to A in order to implement B, the engineer seeks a component x in an expression of the form $A \bullet x = B$, where $\bullet$ is an operator yielding the composite of two design elements. Many compositional theories include the notion of a preorder, usually called refinement. The statement $A \leq C$ usually reads "A refines C" or "A is more specific than C." In this setting, the problem is recast as finding an x such that $A \bullet x \leq B$. It is often assumed that the composition operation is monotonic with respect to this preorder. Therefore, if x is a solution, so is any y satisfying $y \leq x$. This focuses our attention on finding the largest x that satisfies the expression. The literature often calls this largest solution *quotient*.

4.1.1 Background

The logic synthesis community has been a pioneer in defining and solving special cases of the quotient problem for combinational and sequential logic circuit design [36,107] under names like circuit rectification or engineering change or component replacement. In combinational synthesis, much work has been reported to support algebraic and Boolean division: given dividend f and divisor g, find the quotient q and remainder r such $f = q \cdot g + r$ (for $\cdot, +$ standard Boolean operators AND and OR, respectively), as key operation to restructure multi-level Boolean networks [69]. The quotient problem for combinational circuits was formulated as a general replacement problem in [29]: given the combinational circuits A and C whose synchronous composition produces the circuit specification B, what are the legal replacements of C that are consistent with the input-output relation of B? The valid replacements for C were defined as the combinational circuits x such that $A \circ x \subseteq B$, and the largest solution for x was characterized by the closed formula $x = \left(A \circ B^{\perp}\right)^{\perp}$, where $(\cdot)^{\perp}$ is a unary operator that complements the input-output relation of the circuit to which it is applied (switching the inputs and outputs), while a hiding operation gets rid of the internal signals.

In sequential optimization, the typical question addressed was, given a finite-state machine (FSM) A, find an FSM x such that their synchronous composition produces an FSM behaviorally equivalent to a specification FSM B, i.e., solve over FSMs the equation $A \circ x = B$, where $\circ$ is synchronous composition and equality is FSM input-output equivalence. Various topologies were solved, starting with serial composition where the unknown was either the head or tail machine, to more complex interconnections with feedback. As a matter of fact, sometimes both A and x were known, but the goal was to change them into FSMs yielding better logical implementations, while preserving their composition, with the objective to optimize a sequential circuit by computing and exploiting the flexibility due to its modular structure and its environment (see [69,82,181]). An alternative formulation of FSM network synthesis was provided by encoding the problem in the logic WS1S (Weak Second-Order Logic of 1 Successor), which enables to characterize the set of permissible behaviors at a node of a given network of FSMs by WS1S formulas [10], corresponding to regular languages and so to effective operations on finite state automata. [1]

Another stream of contributions has been motivated by component-based design of parallel systems with an interleaving semantics (denoted in our exposition by the composition operator $\diamond$). The problem is stated by Merlin and Bochmann [147] as follows: "Given a complete specification of a given module and the specifications of some submodules, the method described below provides the specification of an additional submodule that, together with the other submodules, will provide a system that satisfies the specification of the given module." The problem was reduced to solving equations or inequalities over process languages, which are usually prefix-closed regular languages represented by labeled transition systems. A closed-form solution of the inequality $A \diamond x \subseteq B$ over prefix-closed regular languages,

[1] A detailed survey of previous work in this area can be found in [102,197].

written as $proj_x(A \diamond B) - proj_x(A \diamond \overline{B})$ (where $proj_x$ is a projection over the alphabet of x), was given in [78, 147].[2] This approach to solve the equation $A \diamond x = B$ has been further extended to obtain restricted solutions that satisfy properties such as safety and liveness, or are restricted to be FSM languages, which need to be input-progressive and avoid divergence (see [26, 78, 197]). The quotient problem has been investigated also for delay-insensitive processes to model asynchronous sequential circuits, see [39, 143, 154]. Equations of the form $A \diamond x \leq B$ were defined, and their largest closed-form solutions were written as $x = (A \diamond B^{\sim})^{\sim}$, where $(\cdot)^{\sim}$ is a suitable unary operation.

An important application from discrete control theory is the model matching problem: design a controller whose composition with a plant matches a given specification (see [14, 58]). Another significant application of the quotient computation has been the protocol design problem, and in particular, the protocol conversion problem (see [34, 75, 79, 110, 113, 160, 163, 199]). Protocol converter synthesis has been studied also over a variant of Input/Output Automata (IOA, [136]), called Interface Automata (IA, [53, 54]), yielding a similar quotient equation $A \diamond_{IA} x \subseteq B$ and closed-form solution $\left(A \diamond_{IA} B^{\perp}\right)^{\perp}$, where $\diamond_{IA}$ is an appropriate interleaving composition defined for interface automata, and $(\cdot)^{\perp}$ is again a unary operation [21].

Some research focused on modal specifications represented by automata whose transitions are typed with *may* and *must* modalities, as in [117, 170], with a solution of the quotient problem for nondeterministic automata provided in [17]. It is outside the scope of this chapter to address the quotient problem for real-time and hybrid systems (see [27, 33] for verification and control in such settings).

As seen above, the quotient problem was studied by different research communities working on various application domains and formalisms. Often similar formulations and solutions were reached albeit obfuscated by the different notations and objectives of the synthesis process. This motivated a concentrated effort to distill the core of the problem, modeling it as solving equations over languages of the form $A \parallel x \preceq B$, where A and B are known components and x is unknown, $\parallel$ is a composition operator, and $\preceq$ is a conformance relation (see [196] and the monograph [197] for full accounts). The notion of language was chosen as the most basic formalism to specify the components of the equation, and language containment $\subseteq$ was selected as conformance relation. Two basic composition operators were defined each encapsulating a family of variants: synchronous composition ($\bullet$) modeling the classical step-lock coordination, and interleaving composition ($\diamond$) modeling asynchrony by which components may progress at different rates (there are subtle issues in comparing the two types, as mentioned in [112, 206]). Therefore two language equations were defined: $A \bullet x \subseteq B$ and $A \diamond x \subseteq B$, where the details of the operations to convert alphabets according to the interconnection topologies are hidden in the formula. It turned out that the largest solutions have the same structure, respectively, $\overline{A \bullet \overline{B}}$ and $\overline{A \diamond \overline{B}}$. This led to investigate the algebraic properties required by the composition operators to deliver the previous largest closed-form

[2] For a discussion about the maximality of this solution and for more references, we refer to [197], Sec. 5.2.1.

solutions to unify the two formulas [196]. This effort assumed that the underlying objects were sets, and that their operations were given in terms of set operations. This work, thus, could not account for quotient computations in more complex theories.

As a parallel development, in recent years we have seen the growth of a rigorous theory of system design based on the algebra of contracts (see the monograph [19]). In this theory, a strategic role is played by assume-guarantee (AG) contracts, in which the *missing component problem* arises: when the given components are not capable of discharging the obligations of the requirements, define a quotient operation that computes the contract for a component, so that by its addition to the original set the resulting system fulfills the requirements. The quotient of AG contracts was completely characterized very recently by a closed-form solution proved in [96]. Once again, the syntax of the quotient has the form $(A \parallel B^{-1})^{-1}$ for contracts A and B and standard contract operations.

In summary, even though the concrete models of the components, composition operators, conformance relations and inversion functions vary significantly across chosen models and application domains, the quotient formulas have similar syntax across theories.

4.1.2 Motivation and contributions

The motivation of this chapter is to *propose the underlying mathematical structure common to all these instances of quotient computation to be able to derive directly the solution formula for any equation satisfying the properties of this common structure.*

We show that we can compute the quotient by only assuming the axioms of a *preorder*, enriched with a binary operation of *source multiplication* and a unary *involution* operation. In particular we introduce the new algebraic notion of *preordered heaps* characterized by a condition, called *admissibility*, which guarantees the existence of the solution and yields a closed form for it. Then we show that a number of theories in computer science meet this condition, e.g., Boolean lattices and AG contracts; so for all of them we are able to (re-)derive axiomatically the formulas that compute their related quotients. We also introduce the concept of *sieved heaps* to deal with structures defined over multiple domains, and we show that the equations $A \bullet x \leq B$ admit a solution also over sieved heaps, generalizing the known solutions of equations on languages over multiple alphabets with respect to synchronous and interleaving composition, well studied in the literature.

4.2 Preordered heaps

In this section we introduce an algebraic structure for which the existence of quotients is guaranteed. First we introduce the notation we will use:

- Let P be a set and let $\mu\colon P \times P \to P$ be a binary operation on P. For any element $a \in P$, we let $\mu_a\colon P \to P$ be the function $\mu_a = \mu \circ (a \times \mathrm{id})$, where id is the identity operator and $(a \times \mathrm{id})\colon P \to P^2$ is the unary function $(a \times \mathrm{id})\colon b \mapsto (a, b)$. Similarly, we

let $\mu^a = \mu \circ (\mathrm{id} \times a)$. If we call μ multiplication, μ_a is left multiplication by a, and μ^a is right multiplication by a.

- For any set P, we let the mapping $\mathrm{Flip}\colon P \times P \to P \times P$ be $\mathrm{Flip}(a, b) = (b, a)$ $(a, b \in P)$.
- Consider a set P and a binary relation $\leq$ on P. Then $\leq$ is a preorder if it is reflexive and transitive; i.e., for all a, b and c in P, we have $a \leq a$ (reflexivity) and if $a \leq b$ and $b \leq c$ then $a \leq c$ (transitivity). If a preorder is antisymmetric, ($a \leq b$ and $b \leq a$ implies $a = b$), then it is a partial order.
- Let $(P, \leq)$ be a preorder and let $a, b \in P$. If $a \leq b$ and $b \leq a$, we write $a \simeq b$.
- Let $F\colon P \to P$. We say that F is monotonic or order-preserving if $a \leq b \Rightarrow Fa \leq Fb$ for all $a, b \in P$. Similarly, we say that F is antitone or order-reversing if $a \leq b \Rightarrow Fb \leq Fa$ for all $a, b \in P$.
- Suppose that $L, R\colon P \to P$ are two monotonic maps on P. We say that (L, R) form an adjoint pair, or that L is the left adjoint of R (R is respectively the right adjoint of L), or that the pair (L, R) forms a Galois connection when for all $b, c \in P$, we have $Lb \leq c$ if and only if $b \leq Rc$.
- Let $F, G\colon P \to P$ be functions on a preorder P. We say that $F \leq G$ when $Fa \leq Ga$ for all $a \in P$.

4.2.1 The concept of preordered heap

As we discussed in the introduction, many times in engineering and computer science one encounters expressions of the form $A \bullet x \leq B$, and one wishes to solve for the largest x that satisfies the expression. The symbols have different specific meanings in the various domains, yet in all applications we know, the syntax for computing the quotient always has the form $\overline{A \bullet \overline{B}}$, where $\overline{(\cdot)}$ is an involution (i.e., a unary operator which is its own inverse). To give meaning to the inequality, at a minimum we need a preorder and a binary operation; to give meaning to the quotient expression, we need to assume the existence of an involution. In all compositional theories, the refinement order has the connotation of specificity: if $a \leq b$ then a is a refinement of b. The binary operation is usually interpreted as composition. The product $a \bullet b$ is understood as the design obtained when operating both a and b in a topology given by the mathematical description of each component. The unary operation is sometimes understood as giving an external view on an object. If a component has mathematical description a, then $\overline{a}$ gives the view that the environment has of the design element. In Boolean algebras, this unary operation is negation. In interface theories, it's usually an operation which switches inputs and output behaviors.

We thus introduce an algebraic structure consisting of a preorder, a binary operation which is monotonic in both arguments, and an involution which is antitone. We have called the binary operation *source multiplication* for reasons having to do with category theory:

we will show that this operation serves as the left functor of an adjunction. Therefore, its application to an object of the preorder yields the *source* of one of the two arrows in the adjunction. Why not simply call it multiplication? Because source multiplication together with the involution generate another binary operation. This second operation we call *target multiplication* because its application to an object yields the *target* of one of the arrows in the adjunction. The unary operation will simply be called *involution*.

The algebraic structure will be called *preordered heap*. The inspiration came from engineering design. In some design methodologies, design elements at the same level of abstraction are not comparable in the refinement order. Indeed, a refinement of a design element usually yields a design element in a more concrete layer. But we are placing all components under the same mathematical structure. This suggested the name *heap*. We add the adjective *preorder* simply to differentiate the concept from existing algebraic heaps. We are ready for the definition:

Definition 4.2.1. *A* preordered heap *is a structure* $(P, \leq, \mu, \gamma)$*, where* $(P, \leq)$ *is a preorder;* $\mu\colon P \times P \to P$ *is a binary operation on* P*, monotonic in both arguments, called* source multiplication*; and* $\gamma\colon P \to P$ *is an antitone operation on* P *called* involution*. These operations satisfy the following axioms:*

- *A1:* $\gamma^2 = id$*.*
- *A2a (left admissibility):* $\mu_a \circ \gamma \circ \mu^a \circ \gamma \leq id \qquad (a \in P)$*.*
- *A2b (right admissibility):* $\mu^a \circ \gamma \circ \mu_a \circ \gamma \leq id \qquad (a \in P)$*.*

Note 4.2.1. *In Definition 4.2.1, we did not assume commutativity in* μ*. If* μ *is commutative, we have* $\mu = \mu \circ Flip$*, so* $\mu_a = \mu \circ (a \times id) = \mu \circ Flip \circ (a \times id) = \mu \circ (id \times a) = \mu^a$*. It follows that for a commutative preordered heap, axioms A2a and A2b become*

$$(\mu_a \circ \gamma)^2 \leq id. \tag{4.1}$$

We have discussed all elements in the definition of a preordered heap, except for the admissibility conditions. What are they? Consider left admissibility: $\mu_a \circ \gamma \circ \mu^a \circ \gamma \leq \mathrm{id}$. Let $b \in P$ and set $B = (\gamma \circ \mu^a \circ \gamma)(b)$. Left admissibility means that B satisfies the expression $\mu(a, x) \leq b$. Similarly, set $C = (\gamma \circ \mu_a \circ \gamma)(b)$. Right admissibility means that C satisfies $\mu(x, a) \leq b$. When μ is commutative, we of course have $B = C$. We will soon show a surprising fact: the axioms of a preordered heap are sufficient to guarantee that B and C are in fact the largest solutions to both expressions, i.e., B and C are the quotients for left and right source multiplication, respectively. We show this immediately after introducing an important binary operation called target multiplication, but first we consider an example.

Example 4.2.2. *Consider a Boolean lattice* B*. The lattice is clearly a preorder. Take the involution to be the negation operator. This is an antitone operator and satisfies A1:* $\neg\neg b = b$ *for all* $b \in B$*. Take source multiplication to be the meet of the lattice (i.e., logical*

AND). This operation is monotonic in the preorder. Since this source multiplication is commutative, the admissibility conditions reduce to checking (4.1)*. For $a, b \in B$, we have $(\mu_a \circ \gamma)^2 b = a \wedge \neg(a \wedge \neg b) = a \wedge (\neg a \vee b) = a \wedge b \leq b$. Thus, the Boolean lattice satisfies the admissibility conditions, making it a preordered heap.* □

4.2.2 Target multiplication

For the rest of this section, let $(P, \leq, \mu, \gamma)$ be a preordered heap. We define the *target multiplication* $\tau \colon P \times P \to P$ as $\tau = \gamma \circ \mu \circ (\gamma \times \gamma)$. Since $\gamma^2 = \mathrm{id}$ (axiom A1), we can also write $\mu = \gamma \circ \tau \circ (\gamma \times \gamma)$, i.e., the diagram $\begin{smallmatrix} P \times P & \xrightarrow{\mu} & P \\ {\scriptstyle \gamma\times\gamma}\updownarrow & & \updownarrow{\scriptstyle \gamma} \\ P \times P & \xrightarrow{\tau} & P \end{smallmatrix}$ commutes.

We could have defined a preordered heap in terms of target multiplication instead of source multiplication. The two operations are closely linked. In fact, we will see in the next section that these operations form an adjoint pair.

Example 4.2.3. *We showed that Boolean lattices are preordered heaps. For B a Boolean lattice and $a, b \in B$, we have $\tau(a, b) = \gamma \circ \mu(\gamma a, \gamma b) = \neg(\neg a \wedge \neg b) = a \vee b$. This suggests that the relation between source and target multiplications is a generalization of De Morgan's identities for Boolean algebras.* □

We will use the following identities: for $a \in P$,

$$\begin{aligned} \mu_a &= \gamma \circ \tau \circ (\gamma \times \gamma) \circ (a \times \mathrm{id}) = \gamma \circ \tau \circ (\gamma a \times \mathrm{id}) \circ \gamma = \gamma \circ \tau_{\gamma a} \circ \gamma \qquad \text{and} \\ \mu^a &= \gamma \circ \tau \circ (\gamma \times \gamma) \circ (\mathrm{id} \times a) = \gamma \circ \tau \circ (\mathrm{id} \times \gamma a) \circ \gamma = \gamma \circ \tau^{\gamma a} \circ \gamma. \end{aligned} \tag{4.2}$$

4.2.3 Solving inequalities in preordered heaps

For $a, b \in P$, we are interested in the conditions under which we can find the largest $x \in P$ such that $\mu(a, x) \leq b$. The following theorem says that source multiplication in a preordered heap is "invertible."

Theorem 4.2.4. *Let $(P, \leq, \mu, \gamma)$ be a preordered heap and let τ be its target multiplication. Then for $a \in P$, $(\mu_a, \tau^{\gamma a})$ and $(\mu^a, \tau_{\gamma a})$ are adjoint pairs.*

Proof. Let $b, c \in P$ with $b \leq \tau^{\gamma a}(c)$. We have $\mu_a(b) \leq (\mu_a \circ \tau^{\gamma a})(c) = (\mu_a \circ \gamma \circ \mu^a \circ \gamma)(c) \leq c$, by left admissibility (by A2a).

Conversely, assume that $\mu_a(b) \leq c$. Then

$$\begin{aligned} \mu_a \circ \gamma^2(b) &\leq c && \text{(by A1)} \\ \gamma \circ (\mu_a \circ \gamma)(\gamma b) &\geq \gamma(c) \\ (\mu^a \circ \gamma) \circ (\mu_a \circ \gamma)(\gamma b) &\geq (\mu^a \circ \gamma)(c) \\ (\gamma b) &\geq (\mu^a \circ \gamma)(c) && \text{(by A2b)} \\ b &\leq (\gamma \circ \mu^a \circ \gamma)(c) = \tau^{\gamma a}(c). && \text{(by A1)} \end{aligned}$$

The adjointness of $(\mu^a, \tau_{\gamma a})$ follows from a similar reasoning. □

The fact that $(\mu_a, \tau^{\gamma a})$ is an adjoint pair means that left source multiplication by a is "inverted" by right target multiplication by γa, i.e.,

$$\mu(a, x) \leq b \quad \text{if and only if} \quad x \leq \tau(b, \gamma a).$$

In other words, the largest solution of $\mu(a, x) \leq b$ is $x = \tau(b, \gamma a)$. Using the familiar multiplicative notation for source multiplication, and $(\cdot)/a = \tau^{\gamma a}$ for "right division by a," we have shown that the largest solution of $ax \leq b$ is $x = b/a$. Calling $a\backslash(\cdot) = \tau_{\gamma a}$ "left division by a," we have shown that the largest solution of $xa \leq b$ is $x = a\backslash b$. These two divisions are related as follows:

Corollary 4.2.5 (Isolating the unknown)**.** *Let P be a preordered heap and $a, x, y \in P$. Then $y \leq a/x$ if and only if $x \leq y\backslash a$.*

Proof. By two applications of Theorem 4.2.4, we obtain $y \leq a/x = \tau^{\gamma x}(a) \Leftrightarrow \mu(x, y) \leq a \Leftrightarrow x \leq \tau_{\gamma y}(a) = y\backslash a$. □

Theorem 4.2.4 is our main result. It shows that preordered heaps have sufficient structure for the computation of quotients. When we prove that a structure is a preordered heap, this theorem immediately yields the existence of an adjoint for multiplication, and its closed form.

In general, to show that a theory is a preordered heap, we must identify its involution and source multiplication. Then we have to verify the admissibility conditions. How difficult is that? Our original problem was identifying the largest x satisfying $\mu(a, x) \leq b$ for some notion of multiplication μ, involution γ, and preorder $\leq$. As we discussed, left admissibility requires that $\tau^{\gamma a} b$ satisfies the inequality $\mu(a, x) \leq b$, and right admissibility requires that $\tau_{\gamma a} b$ satisfies $\mu(x, a) \leq b$. What the theorem tells us is that they are *the largest solutions* to $\mu(a, x) \leq b$ and $\mu(x, a) \leq b$, respectively. In other words, the theorem saves us the effort of making an argument for the optimality of the solutions.

Theorem 4.2.4 also suggests the following observation. For a given $a \in P$, we have adjoint pairs $(\mu_a, \tau^{\gamma a})$ and $(\mu^a, \tau_{\gamma a})$. As we noticed, this means we can find the largest x such that $\mu(a, x) \leq b$ or $\mu(x, a) \leq b$. But it also means that we can find *the smallest* x such that $b \leq \tau(a, x)$ or $b \leq \tau(x, a)$. This is because, $\mu_{\gamma a}$ is the left adjoint of τ^a, and $\mu^{\gamma a}$ is the left adjoint of τ_a. For all examples we will discuss, source multiplication plays the role of the usual composition operation of the theory. But preordered heaps make it clear that μ and τ are closely related operations. In fact, preordered heaps generalize De Morgan's identities (see Section 4.2.2). Thus, while inequalities of the form $\mu(a, x) \leq b$ are more common in the literature, preordered heaps indicate that we can also solve inequalities of the form $b \leq \tau(a, x)$. As we will see, for some theories there is clear understanding of how target multiplication can be used, but for others its use is unknown.

Example 4.2.6. *In the case of a Boolean lattice B, what is the quotient? We showed in previous examples that B is a preordered heap, and we identified its target multiplication. For $a, b \in B$, we can write an expression of the form $\mu(a, x) \leq b$. By Theorem 4.2.4, we know the largest x that satisfies this expression is $\tau^{\gamma a} b = \tau(b, \neg a) = b \vee \neg a$, i.e., the quotient is the implication $a \to b$.* □

4.2.4 Preordered heaps with identity

In the definition of a preordered heap, we did not assume that source multiplication has an identity. Here we consider briefly what happens when it does. Multiplicative identities are common, and in fact, there exists a multiplicative identity in all compositional theories we know.

Suppose P is a preordered heap and $e \in P$ is a left identity for source multiplication, i.e., $\mu_e \simeq \mathrm{id}$. By Theorem 4.2.4, $(\mathrm{id}, \tau^{\gamma e})$ is an adjoint pair. The right adjoint of id is id. Since adjoints are unique up to isomorphism, $\tau^{\gamma e} \simeq \mathrm{id}$. This means that γe is a right identity element for τ. Moreover, in view of (4.2), $\tau_{\gamma e} \simeq \mathrm{id}$. By Theorem 4.2.4, (μ^e, id) is an adjoint pair. By the same reasoning just followed, we must have $\mu^e \simeq \mathrm{id}$. We record this result:

Corollary 4.2.7. *Let $(P, \leq, \mu, \gamma)$ be a preordered heap. If $e \in P$ is a left (or right) identity for source multiplication, it is a double-sided identity for source multiplication, and γe is a double-sided identity for target multiplication. Analogously, if $e \in P$ is a left (or right) identity for target multiplication, it is a double-sided identity for target multiplication, and γe is a double-sided identity for source multiplication.*

Example 4.2.8. *Let B be a Boolean lattice. The top element of the lattice, usually denoted 1, is an identity for source multiplication: $1 \wedge a = a$ for all $a \in B$. The previous corollary tells us that $\neg 1 = 0$ is a double sided identity for target multiplication, which we identified to be disjunction.* □

4.3 Additional instances of preordered heaps

As described in Section 4.2, as soon as we verify that a theory is a preordered heap, we know how to compute quotients for that theory. Here we show that assume-guarantee (AG) contracts are preordered heaps. We first define the algebraic aspects of the theory, and then we proceed to show that it is a preordered heap, which involves verifying the axioms of Definition 4.2.1. After we do this, we invoke Theorem 4.2.4 to express its quotient in closed-form. We limit ourself to defining the theory of assume-guarantee contracts. To learn about their uses and the design methodologies based on them, we suggest [19].

4.3.1 AG contracts

Assume-guarantee contracts are an algebra and a methodology to support compositional system design and analysis. Fix once and for all a set B whose elements we call behaviors. Subsets of B are referred to as behavioral properties or trace properties. An AG contract is a pair of properties $C = (A, G)$ satisfying $A \cup G = B$. Contracts are used as specifications: a component adheres to contract C if it meets the guarantees G when instantiated in an environment that satisfies the assumptions A. The specific form of these properties is not our concern now; we are only interested in the algebraic definitions. The algebra of assume-guarantee contracts was introduced by R. Negulescu [154] (there called *process spaces*) to deal with assume-guarantee reasoning for concurrent programs. The algebra was reintroduced, together with a methodology for system design, by Benveniste et al. [18] to apply assume-guarantee reasoning to the design and analysis of any engineered system. Now we describe the operations of this algebra.

For $C' = (A', G')$ another contract, the partial order of AG contracts, called *refinement*, is given by $C \leq C'$ when $G \subseteq G'$ and $A \supseteq A'$. The involution of AG contracts, called reciprocal, is given by $\gamma C = (G, A)$. This operation is clearly antitone and meets axiom A1. Source multiplication is contract composition: $\mu(C, C') = (A \cap A' \cup \neg(G \cap G'), G \cap G')$. This operation yields the tightest contract obeyed by the composition of two design elements, each obeying contracts C and C', respectively. Composition is monotonic in the refinement order of AG contracts. We need to verify the admissibility conditions. Since source multiplication for AG contracts is commutative, we verify (4.1):

$$\begin{aligned}(\mu_C \circ \gamma)^2 C' &= (\mu_C \circ \gamma) \circ (\mu_C)(G', A') = \mu_C(G \cap A', A \cap G' \cup \neg(G \cap A')) \\ &= (A \cap G \cap A' \cup \neg G \cup \neg(A \cap G' \cup \neg A'), G \cap (A \cap G' \cup \neg A')) \\ &= (A \cap A' \cup \neg G \cup \neg A \cap A' \cup \neg G' \cap A', G \cap (A \cap G' \cup \neg A')) \\ &= (A' \cup \neg G, G \cap (A \cap G' \cup \neg A')) \leq (A', G') = C',\end{aligned}$$

where in the last step we used the fact that $\neg A' \subseteq G'$, which follows from $A' \cup G' = B$. We conclude that AG contracts satisfy the admissibility conditions, and thus have preordered heap structure.

What is target multiplication for AG contracts? From its definition, we have $\tau(C, C') = \gamma \circ \mu \circ (\gamma C, \gamma C') = \gamma \circ \mu((G, A), (G', A')) = (A \cap A', G \cap G' \cup \neg(A \cap A'))$. This is an operation on contracts called *merging*. One of the main objectives of the theory of assume-guarantee contracts is to deal with *multiple viewpoints*, i.e., a multiplicity of design concerns, each having a contract representing the specification for that concern (e.g., functionality, timing, etc.). In [161], it is argued that the operation of merging is used to bring multiple viewpoint specifications into a single contract object.

Since AG contracts are preordered heaps, we get their quotient formulas from Theorem 4.2.4. The adjoint of $\mu_{C'}$ is $\tau^{\gamma C'} = \gamma \circ \mu^{C'} \circ \gamma$. Applying this to C yields $\tau^{\gamma C'}(C) = \gamma \circ \mu^{C'}(G, A) = (A \cap G', G \cap A' \cup \neg(A \cap G'))$. This closed-form expression for the quotient of AG contracts was first reported in [96]. Also by Theorem 4.2.4, the left adjoint of merging by a fixed contract C' is the operation $\mu(C, \gamma C') = \mu((A, G), (G', A')) =$

$(A \cap G' \cup \neg(G \cap A'), G \cap A')$. This operation was recently introduced under the name of *separation* in [161].

4.4 Sieved heaps

Some theories in computer science require manipulating objects which are not defined over the same domain. For example, consider a language L_1 defined over an alphabet Σ_1. Let Σ_2 be another alphabet for which L_2 is a language. The powerset of a set is a Boolean lattice, so we have two preordered heaps $P_{\Sigma_1} = 2^{\Sigma_1^*}$ and $P_{\Sigma_2} = 2^{\Sigma_2^*}$ whose source multiplications and involutions are intersection and negation ($*$ is the Kleene star—we will define operations carefully in the section on languages). With the theory of preordered heaps, we know how to solve inequalities for P_{Σ_1} and for P_{Σ_2}. Suppose we define an operation that allows us to compose $L_1 \in P_{\Sigma_1}$ with $L_2 \in P_{\Sigma_2}$. How do we solve inequalities involving L_1 and L_2 then? These languages belong to different preordered heaps. It is natural to define such an operation by mapping L_1 and L_2 to a common preordered heap, which by definition, has its own notion of source multiplication. We need a notion of mapping between preordered heaps:

Definition 4.4.1. *Let $(P, \leq, \mu, \gamma)$ and $(P', \leq', \mu', \gamma')$ be two preordered heaps. A preordered heap homomorphism $f\colon P \to P'$ is an order-preserving map which commutes with the source multiplications and involutions, i.e.,* $\begin{array}{ccc} P \times P & \xrightarrow{f \times f} & P' \times P' \\ {\scriptstyle \mu}\downarrow & & \downarrow{\scriptstyle \mu'} \\ P & \xrightarrow{f} & P' \end{array}$ *and* $\begin{array}{ccc} P & \xrightarrow{f} & P' \\ {\scriptstyle \gamma}\downarrow & & \downarrow{\scriptstyle \gamma'} \\ P & \xrightarrow{f} & P' \end{array}$ *commute.*

Preordered heaps P_{Σ_1} and P_{Σ_2} are indexed by alphabets. The common preordered heap where L_1 and L_2 can be mapped is determined by Σ_1 and Σ_2. As we will see in the next section, one option is to say that they generate the alphabet $\Sigma_c = \Sigma_1 \cup \Sigma_2$, and we can define maps $\iota_1\colon P_{\Sigma_1} \to P_{\Sigma_c}$ and $\iota_2\colon P_{\Sigma_2} \to P_{\Sigma_c}$ that embed languages over Σ_1 and Σ_2 to those defined under Σ_c. This observation tells us that we can use a structure S in order to index preordered heaps; this structure must have a binary operation defined in it. This operation will fulfill the role of identifying the alphabets where two languages can meet. Call this structure S, and let $\cdot$ be its binary operation. If we have two languages defined over the same alphabet, we should not need to move to another alphabet to compute the source multiplication of the two languages; thus, the binary operation of S should be idempotent. We will also require the operation to be commutative since it makes no difference whether we go to the language generated by Σ_1 and Σ_2 or to that generated by Σ_2 and Σ_1. A similar reasoning leads us to require associativity. Thus, S is endowed with an associative, commutative, idempotent binary operation, which means it is a semilattice. We make the choice to interpret it as an upper semi-lattice because we have the intuition that the languages generated by two smaller languages should be larger than any of the two, but this interpretation does not impose any algebraic limitations: an upper semilattice can be turned into a lower semilattice simply by flipping it upside-down.

We introduce the notion of a sieved, preordered heap (sieved heap, for short) that allows us to move objects between different domains of definition or different levels of abstraction. A sieved heap is a collection of preordered heaps indexed by an upper semilattice S together with mappings between the preordered heaps. We call these mappings concretizations. An upper semilattice can be interpreted as a partial order: for $a, b \in S$, we say that $a \leq ab$. Thus, the shortest definition for a sieved heap is that it is a functor from the preorder category S to **PreHeap**, the preordered heap category, whose objects are preordered heaps and whose arrows are preordered heap homomorphisms. We will give a longer definition. But first, why the adjective sieved? A sieved heap consists of a collection of preordered heaps and maps between them. We interpret these preordered heaps as structures containing varying amounts of detail about an object. This varying granularity motivated the name. This is the definition of this composite structure:

Definition 4.4.2. *Let S be a semilattice. Let $\{(P_x, \leq_x, \mu_x, \gamma_x)\}_{x \in S}$ be a collection of preordered heaps such that for every $x, y, z \in S$ we have a unique preordered heap homomorphism $\iota\colon P_x \to P_{xy}$ referred to as a concretization and making* $\begin{smallmatrix} & & P_{xy} & & \\ & {\scriptstyle\iota}\uparrow & & \searrow^{\iota'} & \\ P_x & & \xrightarrow[\iota'']{} & & P_{xyz} \end{smallmatrix}$ *commute. We require the concretization $\iota\colon P_x \to P_x$ to be the identity. Let $P = \oplus_{x \in S} P_x$, where $\oplus$ stands for disjoint union. We call $(P, \leq, \mu, \gamma)$ an S-sieved heap, where $\mu\colon P \times P \to P$ is an operation called source multiplication, and $\gamma\colon P \to P$ is called involution. Let $a \in P_x$ and $b \in P_y$, and let $\iota_x\colon P_x \to P_{xy}$ and $\iota_y\colon P_y \to P_{xy}$ be concretizations. These operations are given by*

$$\mu(a, b) = \mu_{xy}\left(\iota_x(a), \iota_y(b)\right) \quad \text{and} \quad \gamma(a) = \gamma_x(a).$$

Moreover, we say that $a \leq b$ if and only if there exists $z \in S$ and concretizations $\iota\colon P_x \to P_z$ and $\iota'\colon P_y \to P_z$ such that $\iota(a) \leq_z \iota'(b)$, where $\leq_z$ is the preorder of P_z.

Target multiplication τ for P is defined in a similar way: $\tau(a, b) = \tau_{xy}\left(\iota_x(a), \iota_y(b)\right)$, where τ_{xy} is the target multiplication of the preordered heap P_{xy}.

4.4.1 Sieved heaps are preordered heaps

Now we show that a sieved heap is itself a preordered heap. To do this, we must show that the relation $\leq$ over sieved heaps is a preorder, that source multiplication defined for a sieved heap is monotonic, that its involution is antitone, and that it meets the admissibility conditions. The following statements show that sieved heaps have these properties.

Lemma 4.4.3. *The relation $\leq$ on an S-sieved heap P is a preorder.*

Proof. Reflexivity. Let $a \in P_x$. Let ι be the concretization $\iota\colon P_x \to P_x$. Then $\iota a \leq_x \iota a$ because $\leq_x$ is a preorder in P_x; this means that $a \leq a$ in P.

Transitivity. Let $b \in P_y$ and $c \in P_z$ and suppose that $a \leq b$ and $b \leq c$. Then there exist $v, w \in S$ such that $\iota_x a \leq_v \iota_y b$ and $\iota'_y b \leq_w \iota_z c$, where the diagram $\begin{smallmatrix} P_v & \xrightarrow{\iota_v} & P_{vw} & \xleftarrow{\iota_w} & P_w \\ {\scriptstyle\iota_x}\uparrow & \nwarrow^{\iota_y} & & \nearrow_{\iota'_y} & \uparrow{\scriptstyle\iota_z} \\ P_x & & P_y & & P_z \end{smallmatrix}$ shows

the relevant concretization maps (these diagrams commute per Definition 4.4.2). We obtain immediately $\iota_v \circ \iota_x a \leq_{vw} \iota_v \circ \iota_y b$ and $\iota_w \circ \iota'_y b \leq_{vw} \iota_w \circ \iota_z c$. From the diagram, $\iota_v \circ \iota_y = \iota_w \circ \iota'_y$, which means that $\iota_v \circ \iota_x a \leq_{vw} \iota_w \circ \iota_z c$, which means that $a \leq c$. □

Lemma 4.4.4. *Source multiplication on P is monotonic in both arguments.*

Proof. Let $a, b, c \in P$ with $a \leq c$. Suppose that $a \in P_x$, $b \in P_y$, and $c \in P_z$. Since $a \leq c$, there exist $u \in S$ such that $\iota_x a \leq_u \iota_z c$ for concretizations $\iota_x \colon P_x \to P_u$ and $\iota_z \colon P_z \to P_u$. Note that this means there exist $u', u'' \in S$ such that $u = xu'$ and $u = zu''$. But this implies that $uy = xyu'$ and $uy = yzu''$. Thus, there exist concretizations $\iota_{xy} \colon P_{xy} \to P_{uy}$ and $\iota_{yz} \colon P_{yz} \to P_{uy}$, and

$$\begin{array}{ccccc} & & P_y & & \\ & \swarrow^{\iota'_y} & \downarrow{\scriptstyle \iota_y} & \searrow^{\iota''_y} & \\ P_{xy} & \xrightarrow[\iota_{xy}]{} & P_{uy} & \xleftarrow[\iota_{yz}]{} & P_{yz} \\ {\scriptstyle \iota'_x}\uparrow & & \uparrow{\scriptstyle \iota_u} & & \uparrow{\scriptstyle \iota'_z} \\ P_x & \xrightarrow{\iota_x} & P_u & \xleftarrow{\iota_z} & P_z \end{array} \tag{4.3}$$

commutes. Since $a \leq c$, we have

$$\mu_{uy}\left(\iota_u \circ \iota_x a, \iota_y b\right) \leq_{uy} \mu_{uy}\left(\iota_u \circ \iota_z c, \iota_y b\right). \tag{4.4}$$

By the commutativity of the diagram, $\iota_y = \iota_{xy} \circ \iota'_y = \iota_{yz} \circ \iota''_y$ and $\iota_u \circ \iota_x = \iota_{xy} \circ \iota'_x$ and $\iota_u \circ \iota_z = \iota_{yz} \circ \iota'_z$. Using these identities, we can rewrite (4.4) as

$$\begin{aligned} &\mu_{uy}\left(\iota_{xy} \circ \iota'_x a, \iota_{xy} \circ \iota'_y b\right) \leq_{uy} \mu_{uy}\left(\iota_{yz} \circ \iota'_z c, \iota_{yz} \circ \iota''_y b\right), \quad \text{which implies that} \\ &\quad \iota_{xy} \circ \mu_{xy}\left(\iota'_x a, \iota'_y b\right) \leq_{uy} \iota_{yz} \circ \mu_{yz}\left(\iota'_z c, \iota''_y b\right) \text{ and thus } \iota_{xy} \circ \mu\left(a, b\right) \leq_{uy} \iota_{yz} \circ \mu\left(c, b\right). \end{aligned}$$

This shows that $\mu\left(a, b\right) \leq \mu\left(c, b\right)$. Monotonicity in the second argument is proved in the same way. □

Theorem 4.4.5. *An S-sieved heap P is a preordered heap.*

Proof. By lemma 4.4.3, we know that $(P, \leq)$ is a preorder. By lemma 4.4.4, we know that source multiplication for P is monotonic. From the definition of involution γ for P, it is immediate that this operation is antitone and that $\gamma^2 = \mathrm{id}$. We must show the admissibility conditions. Let $a \in P_x$ and $b \in P_y$. Using the notation of (4.3), we have $\mu(a, \gamma \circ \mu(\gamma b, a)) = \mu(a, \gamma \circ \mu_{xy}(\iota'_y \circ \gamma b, \iota'_x a)) = \mu_{xy}(\iota'_x a, \gamma \circ \mu_{xy}(\gamma \circ \iota'_y b, \iota'_x a)) \leq \iota'_y b$, where we used the left admissibility of the preordered heap P_{xy}. But this means that $\mu(a, \gamma \circ \mu(\gamma b, a)) \leq b$. We conclude that P meets the left admissibility condition. Applying the same procedure tells us that P also has right admissibility. Thus, P is a preordered heap. □

Now that we know that sieved heaps are preordered heaps, we can compute quotients in these structures. We will now consider the solution of inequalities over languages as an application of sieved heaps.

4.5 Sieved heaps and language inequalities

Language inequalities arise as the formalization of the problem of synthesizing an unknown component in hardware and software systems. In this section, we provide preliminaries on languages and discuss their properties and operations. A fuller treatment of language properties can be found in [197, 206]. Our objective is to show that commonly studied language structures are sieved heaps, which allows us to axiomatically find their quotients per the results of Section 4.4.

4.5.1 Operations on languages

An alphabet is a finite set of symbols. The set of all finite strings over a fixed alphabet X is denoted by $X^\star$. $X^\star$ includes the empty string ϵ. A subset $L \subseteq X^\star$ is called a **language** over alphabet X. [90] is a standard reference on this subject.

A **substitution** f is a mapping of an alphabet Σ to subsets of $\Delta^\star$ for some alphabet Δ. The substitution f is extended to strings by setting $f(\epsilon) = \{\epsilon\}$ and $f(xa) = f(x)f(a)$. The following are well-studied language operations.

- Given a language L over alphabet X and an alphabet V, consider the substitution $l\colon X \to 2^{(X\times V)^\star}$ defined as $l(x) = \{(x, v) \mid v \in V\}$. Then the language $L_{\uparrow V} = \cup_{\alpha\in L} l(\alpha)$ over alphabet $X \times V$ is the **lifting** of language L to alphabet V.
- Given a language L over alphabet X and an alphabet V, consider the mapping $e\colon X \to 2^{(X\cup V)^\star}$ defined as $e(x) = \{\alpha x\beta \mid \alpha, \beta \in (V - X)^\star\}$. Then the language $L_{\Uparrow V} = \cup_{\alpha\in L} e(\alpha)$ over alphabet $X \cup V$ is the **expansion** of language L to alphabet V, i.e., words in $L_{\Uparrow V}$ are obtained from those in L by inserting anywhere in them words from $(V - X)^\star$. Notice that e is not a substitution and that $e(\epsilon) = \{\alpha \mid \alpha \in V^\star\}$.

The following proposition states that language liftings and expansions meet the properties of concretization maps of a sieved heap. These results will be used in the next section dealing with inequalities over languages.

Proposition 4.5.1. *Liftings and expansions are order-preserving and commute with intersection and complementation.*

Proof. Let L, L_1, L_2 be languages over alphabet X, we need to show that the following properties hold:

a. $L_1 \subseteq L_2$ implies $L_{1\ \uparrow V} \subseteq L_{2\ \uparrow V}$;

b. $L_1 \subseteq L_2$ implies $L_{1\ \Uparrow V} \subseteq L_{2\ \Uparrow V}$;

c. $(L_1 \cap L_2)_{\uparrow V} = L_{1\ \uparrow V} \cap L_{2\ \uparrow V}$;

d. $(L_1 \cap L_2)_{\Uparrow V} = L_{1\ \Uparrow V} \cap L_{2\ \Uparrow V}$;

e. $\overline{L}_{\uparrow V} = \overline{L_{\uparrow V}}$; and

f. $\overline{L}_{\Uparrow V} = \overline{L_{\Uparrow V}}$.

(a) and (b) are a consequence of the fact that $L_1 \subseteq L_2$ implies $L_1 \cup L_2 = L_2$. Observe that $L_{1\uparrow V} = \cup_{\alpha\in L_1} l(\alpha) \subseteq \cup_{\alpha\in L_1\cup L_2} l(\alpha) = \cup_{\alpha\in L_2} l(\alpha) = L_{2\uparrow V}$ and $L_{1\Uparrow V} = \cup_{\alpha\in L_1} e(\alpha) \subseteq \cup_{\alpha\in L_1\cup L_2} e(\alpha) = \cup_{\alpha\in L_2} e(\alpha) = L_{2\Uparrow V}$.
Parts (c) and (d) are proved by Proposition 2.2 in [206].
(e) $\alpha_1 \neq \alpha_2$ implies $l(\alpha_1) \cap l(\alpha_2) = \emptyset$, then

$$L_{\uparrow V} \cap \overline{L}_{\uparrow V} = (\cup_{\alpha\in L} l(\alpha)) \cap (\cup_{\alpha\in\overline{L}} l(\alpha)) = \emptyset.$$

Moreover, we have $L_{\uparrow V} \cup \overline{L}_{\uparrow V} = (\cup_{\alpha\in L} l(\alpha)) \cup (\cup_{\alpha\in\overline{L}} l(\alpha)) = (X \times V)^\star$. Hence, $\overline{L}_{\uparrow V} = (X \times V)^\star - L_{\uparrow V} = \overline{L_{\uparrow V}}$.
(f) In a similar way, since $\alpha_1 \neq \alpha_2$ implies $e(\alpha_1) \cap e(\alpha_2) = \emptyset$, then

$$L_{\Uparrow V} \cap \overline{L}_{\Uparrow V} = (\cup_{\alpha\in L} e(\alpha)) \cap (\cup_{\alpha\in\overline{L}} e(\alpha)) = \emptyset.$$

We also have $L_{\Uparrow V} \cup \overline{L}_{\Uparrow V} = (\cup_{\alpha\in L} e(\alpha)) \cup (\cup_{\alpha\in\overline{L}} e(\alpha)) = (X \cup V)^\star$. Hence, $\overline{L}_{\Uparrow V} = (X \cup V)^\star - L_{\Uparrow V} = \overline{L_{\Uparrow V}}$. □

4.5.2 Composition of languages and inequalities involving languages

Consider two systems A and B with associated languages $L(A)$ and $L(B)$. The systems communicate with each other by a channel U and with the environment by channels I and O. The following two well-studied operators describe the external behavior of the composition of $L(A)$ and $L(B)$.

Definition 4.5.2. *Given the disjoint alphabets I, U, O, a language L_1 over $I \times U$, and a language L_2 over $U \times O$, the **synchronous composition** of languages L_1 and L_2 is the language $(L_1)_{\uparrow O} \cap (L_2)_{\uparrow I}$, denoted by $L_1 \bullet L_2$, defined over $I \times U \times O$.*

Definition 4.5.3. *Given the disjoint alphabets I, U, O, a language L_1 over $I \cup U$, and a language L_2 over $U \cup O$, the **parallel composition** of languages L_1 and L_2 is the language $(L_1)_{\Uparrow O} \cap (L_2)_{\Uparrow I}$, denoted by $L_1 \diamond L_2$, defined over $I \cup U \cup O$.*

Example. Let $L_1 = \{a, aa\}$ be a language of the alphabet $\Sigma_1 = \{a, b\}$, and $\Sigma_2 = \{c, d\}$ be another alphabet for which $L_2 = \{c\}$ is a language. Then $L_1 \bullet L_2 = \{(a, c)\}$ and $L_1 \diamond L_2 = \{ac, ca, caa, aca, aac\}$.

Synchronous composition abstracts the parallel execution of modules in lock step, assuming a global clock and instant communication by a broadcasting mechanism, modeling the product semantics common in the hardware community. In asynchronous composition modules execute independently at different speeds assuming clocks which progress at arbitrary

rates relative to one another, modeling the interleaving semantics common in the software community. A comparison can be found in [112]. Now we show that we can interpret the above products as the source multiplication of a sieved heap. For each product, we first need to identify a suitable indexing semilattice. Then we need to build the appropriate preordered heaps and their maps.

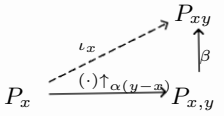
4.5.2.1 Synchronous equations

Semilattice. Suppose we have a disjoint family $F = \{\Sigma_i\}_{1 \leq i \leq n}$ of alphabets for some positive integer n, and let $S = 2^F$. Then S is a semilattice under the operation of set union, i.e., if $x, y \in S$, we have $xy = x \cup y$.

Preordered heaps. For any $x \in S$, let $|x|$ be the cardinality of x. There exist natural numbers $k_1, \ldots, k_{|x|}$ such that $x = \{\Sigma_{k_j}\}_{1 \leq j \leq |x|} \subseteq F$ and $1 \leq k_i < k_j \leq n$ for $i < j$. We map each x to a preordered heap as follows. We define the alphabet over x as $\alpha(x) = \Sigma_{k_1} \times \cdots \times \Sigma_{k_{|x|}}$, and we set $P_x = 2^{\alpha(x)^*}$. Source multiplication μ_x for P_x is intersection, and involution γ_x is complementation. $(P_x, \leq_x, \mu_x, \gamma_x)$ is a Boolean lattice, thus a preordered heap, as shown in Section 4.2.

Concretizations. For $x, y \in S$, P_{xy} is clearly a preordered heap because $xy \in S$. We also define the preordered heap $P_{x,y} = 2^{\Sigma_{x,y}^*}$ for $\Sigma_{x,y} = \alpha(x) \times \alpha(y - x)$ with source multiplication equal to set intersection and involution equal to complementation. Note that the only difference between P_{xy} and $P_{x,y}$ is the order in which the alphabets Σ_i appear in each: P_{xy} contains all sets of finite strings over the alphabet $\alpha(xy)$, and $P_{x,y}$ contains all sets of finite strings over the alphabet $\alpha(x) \times \alpha(y - x)$. Thus, P_{xy} and $P_{x,y}$ are isomorphic as sets. Let $\beta \colon P_{x,y} \to P_{xy}$ be this isomorphism, which is easily seen to be a preordered heap isomorphism.

This allows us to define the concretization ι_x as follows:



.

From Proposition 4.5.1, we know that $(\cdot) \uparrow_{\alpha(y-x)}$ is a preordered heap map. Thus, we have an S-sieved heap $\{(P_x, \leq_x, \mu_x, \gamma_x)\}_{x \in S}$. Since sieved heaps are preordered heaps (Theorem 4.4.5), for $A \in P_x$ and $B \in P_y$, an equation of the form $A \bullet z \leq B$ has the largest solution $Z \in P_{xy}$ with

$$Z = \neg \left(\neg \beta' \left(B \uparrow_{\alpha(x-y)} \right) \cap \ \beta'' \left(A \uparrow_{\alpha(y-x)} \right) \right),$$

where $\beta' \colon P_{y,x} \to P_{xy}$ and $\beta'' \colon P_{x,y} \to P_{xy}$ are extensions of the alphabet permutations to languages, as described above.

Example 4.5.4. *Let I, U, and O be disjoint alphabets. Then S consists of all subsets of $\{I, O, U\}$. Let $i = \{I\}$, $u = \{U\}$, and $o = \{O\}$. The preordered heap P_{iu} consists of all languages over the alphabet $I \times U$. P_{uo} consists of all languages over $U \times O$. If $L_1 \in P_{iu}$, the concretization $\iota \colon P_{iu} \to p_{iuo}$ maps L_1 to a language over $I \times U \times O$. Observe that the order in which each alphabet appears is important and set from the beginning; this eliminates any potential ambiguities with the ordering of the alphabets (e.g., is it the alphabet $I \times U$ or $U \times I$?). By definition, this concretization map is $(\cdot) \uparrow_O$. In the same way, the concretization*

$\iota' \colon P_{uo} \to p_{iuo}$ is $\beta \circ (\cdot) \uparrow_I$, where $\beta : P_{uo,i} \to P_{iuo}$ permutes the symbols of the language so that they appear in the order (a, b, c) with $a \in I$, $b \in U$, and $c \in O$. Thus, source multiplication is $\mu(L_1, L_2) = L_1 \uparrow_O \cap \beta\,(L_2 \uparrow_I)$, which is the synchronous product. □

4.5.2.2 Asynchronous equations

Now we form a semilattice S whose elements are abstract sets and whose operation is set union. Let $x \in S$, and define $P_x = 2^{x^*}$. For $y \in S$, the concretization $P_x \xrightarrow{\iota} P_{xy}$ is $\iota = (\cdot) \Uparrow_{y-x}$. Proposition 4.5.1 shows that ι is a preordered heap map. Thus, we have a sieved heap $\{(P_x, \leq_x, \mu_x, \gamma_x)\}_{x \in S}$.

Since sieved heaps are preordered heaps (Theorem 4.4.5), we are in a position to solve language equations under asynchronous composition. Let $x, y \in S$, $A \in P_x$ and $B \in P_y$. The largest solution to the equation $A \diamond z \leq B$ yields $Z \in P_{xy}$ with $Z = \neg\,(\neg B \Uparrow_{x-y} \cap\, A \Uparrow_{y-x})$.

Example 4.5.5. *As before, let I, U, and O be disjoint alphabets, and let $I, U, O \in S$, where S is a semilattice with the operation of set union. The preordered heap P_{IU} consists of all languages over $I \cup U$. Similarly, the preordered heap P_{UO} consists of all languages over $U \cup O$. The embedding $\iota \colon P_{IU} \to P_{IUO}$ is simply $(\cdot) \Uparrow_O$, and the embedding $\iota' \colon P_{UO} \to P_{IUO}$ is $(\cdot) \Uparrow_I$. Thus, for $L_1 \in P_{IU}$ and $L_2 \in P_{UO}$, source multiplication is $\mu(L_1, L_2) = L_1 \Uparrow_O \cap L_2 \Uparrow_I$, which is the asynchronous product.* □

4.6 Summary

The comparison of the closed form computation of quotients ranging from language equations to AG contracts suggested a new algebraic structure, called *preordered heap*, endowed with the axioms of preorders, together with a monotonic multiplication and an involution. We showed that an admissibility condition allows to solve equations over preordered heaps, and we gave the closed form of the solution. We showed that various theories qualify as preordered heaps and therefore admit such explicit solution. In particular, we showed that the conditions for being preordered heaps hold for Boolean lattices and assume-guarantee contracts: in both cases we were able to derive axiomatically the quotients, which had been previously obtained by specific analysis of each theory. Finally we defined equations over sieved heaps to handle components defined over multiple alphabets, and rederived as special cases the solution of language equations known in the literature.

Chapter 5

Contract merging and separation

Besides parallel composition, there is another binary operation we would like to carry out on pairs of contracts. Assume two contracts specify different aspects of the same component. One contract could specify, for example, how the component behaves functionally, and another could describe its timing or power characteristics. We call these aspects *viewpoints*. The operation of *viewpoint merging* consists in combining various contracts describing different aspects of the same component into a single contract object. A natural choice for merging, and one that is followed by most of the literature on contracts, is the conjunction operation [18, 19]. We here show, however, that conjunction is not always the answer to viewpoint merging.

5.1 A revised notion of contract merging

Suppose a device guarantees to output data at certain rate R_o provided the data rate r_i of the input is higher than some minimum R_L. Suppose the same device guarantees it will consume less than P units of power if the temperature is bounded above by T_H. The contract rejects environments that do not satisfy such requirements. We can write the following functional and power contracts for this device:

$$\mathcal{C}_F = (r_i > R_L, r_o = R_o) \quad \text{and} \quad \mathcal{C}_P = (T < T_H, p < P).$$

We can take their conjunction after inverse projecting to equalize the alphabets. From a syntactic standpoint, inverse projection has no impact on the definition of the contract: if a variable is added to the requirements or guarantees, the proposition added must allow the variable to take any value in its domain, which is a true proposition. For example, the requirements of $\mathcal{C}_F$ can be extended with the proposition $0 < T < \infty$, which is always true since T is a non-negative real number. The extended requirements are shown graphically in Figure 5.1.a and 5.1.b. Before carrying out conjunction, it is necessary to saturate both

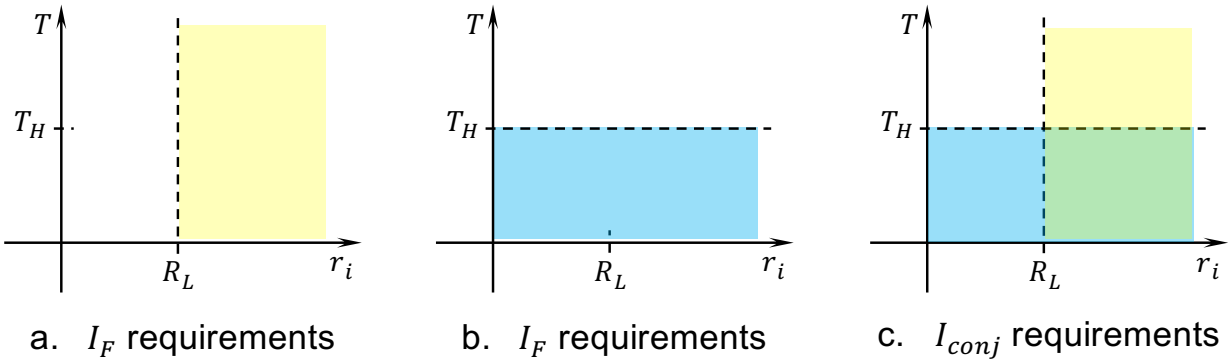


Figure 5.1: Requirements: rate, temperature, and their union

contracts. We obtain

$$\mathcal{C}_F = (r_i > R_L,\ r_o = R_o \vee \neg(r_i > R_L)) \quad \text{and}$$
$$\mathcal{C}_P = (T < T_H,\ p < P \vee \neg(T < T_H)).$$

The conjunction is given by

$$\begin{aligned}\mathcal{C}_{\text{conj}} = (&r_i > R_L \vee T < T_H\ ,\\ &(r_o = R_o \wedge p < P) \vee (r_o = R_o \wedge T \geq T_H) \vee\\ &(p < P \wedge r_i \leq R_L) \vee (T \geq T_H \wedge r_i \leq R_L).\end{aligned}$$

The resulting contract allows a satisfying component to guarantee either the guarantees of both viewpoints (i.e., $r_o = R_o \wedge p < P$) or the guarantees of only one of the viewpoints when the assumptions of only one of them holds (e.g., $r_o = R_o \wedge T \geq T_H$) or to guarantee nothing when none of the assumptions hold (i.e., $T \geq T_H \wedge r_i \leq R_L$). The requirements of the conjunction, shown in Figure 5.1.c, are computed as the union and include the entire shaded area. These requirements appear too permissive, as a satisfying component must now be able to accept environments that produce rates below R_L, and must work at temperatures potentially higher than T_H. The problem lies in the *inverse projection combined with the union*. Ideally, we would like instead to consider only the intersection, which corresponds to only the green area in Figure 5.1.c. This must be regarded as a new operator, since the form of conjunction depends on the refinement order, and cannot simply be redefined.

The idea is that viewpoint merging should tell us what all viewpoints guarantee simultaneously since *the contract for each viewpoint demands that its assumptions be met*. Thus, contract *merging* is defined as product with guarantee relaxation:

Definition 5.1.1 (Merging)**.** *Let $\mathcal{C}_1$ and $\mathcal{C}_2$ be contracts. Then $\mathcal{C}$ is the result of merging $\mathcal{C}_1$ and $\mathcal{C}_2$, written $\mathcal{C} = \mathcal{C}_1 \bullet \mathcal{C}_2$, if and only if*

$$G = (G_1 \cap G_2) \cup \neg\,(A_1 \cap A_2) \quad \textit{and} \quad A = A_1 \cap A_2.$$

Applying the merging operator to our example yields:

$$\begin{aligned}\mathcal{C}_F \bullet \mathcal{C}_P = &(r_i > R_L \wedge T < T_H, (r_o = R_o \vee \neg(r_i > R_L)) \wedge \\ &\quad (p < P \vee \neg(T < T_H)) \vee r_i \leq R_L \vee T \geq T_H) \\ = &(R_L < r_i \wedge T < T_H, \\ &\quad (r_o = R_o \wedge p < P) \vee T \geq T_H \vee r_i \leq R_L)\,.\end{aligned}$$

The guarantees of $\mathcal{C}_F \bullet \mathcal{C}_P$ now require that the guarantees of both contracts hold; moreover, this contract forces the environment to meet the requirements of both contracts. Thus, the merging operation correctly captures the intuitive notion of viewpoint merging. In what follows, we discuss the properties of the operator and how it fits coherently in the overall contract theory.

5.2 Composition, merging, and the contract lattice

At this point, contracts have two binary operations introduced by axiom and with semantic meaning: composition and merging. The contract model, however, is a partial order on refinement with well defined lattice operations. How are merging and composition related to the contract lattice? The following theorem tells us that merging is one of the components of the GLB.

Theorem 5.2.1. *Let $\mathcal{C} = (A, G)$ and $\mathcal{C}' = (A', G')$ be contracts. Then the contract $\mathcal{C} \wedge \mathcal{C}'$ is equal to the conjunction of the following three contracts:*

1. $(A - A', G \cup \neg\,(A - A'))$
2. $(A' - A, G' \cup \neg\,(A' - A))$
3. $(A \cap A', G \cap G' \cup \neg\,(A \cap A'))$

Proof. The union of the requirements of the three contracts clearly yields $A \cup A'$. We compute the intersection of the guarantees:

$$\begin{aligned}&(G \cup A') \cap (G' \cup A) \cap (G \cap G' \cup \neg\,(A \cap A')) \\ &\quad = (G \cup A') \cap ((G \cap G') \cup (G' \cap \neg A) \cup (G' \cap \neg A') \cup (A \cap \neg A')) \\ &\quad = (G \cup A') \cap ((G \cap G') \cup \neg A') \quad (\text{since } G \cap \neg A' \leq G \cap G') \\ &\quad = ((G \cap G') \cup (G \cap \neg A')) = G \cap G',\end{aligned}$$

which are the guarantees of the conjunction. □

We think of Theorem 5.2.1 as providing a factorization of the conjunction of two contracts into three contracts. The first two contracts of this factorization require the environment to support the environments of only one of $\mathcal{C}$ or $\mathcal{C}'$, and provide the guarantees only of one of

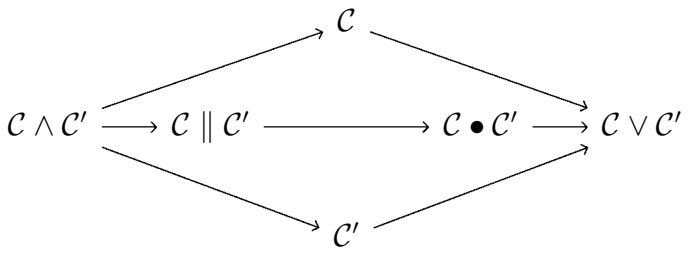


Figure 5.2: Given contracts $\mathcal{C}$ and $\mathcal{C}'$, their operations of conjunction, disjunction, composition, and merging are ordered.

the contracts. The third contract, on the other hand, which corresponds to merging, requires the environment to support the assumptions of both contracts simultaneously, and provides the guarantees of both contracts. The use of the first two contracts in the factorization is likely to be limited. The third contract, however, represents what the component does *when both viewpoints being conjoined are active simultaneously.* As we discussed, this corresponds exactly to the notion of *viewpoint merging*, hence the name of the operation.

We now observe that composition is, dually to merging, a component of the factorization of the disjunction of two contracts.

Theorem 5.2.2. *Let $\mathcal{C} = (A, G)$ and $\mathcal{C}' = (A', G')$ be contracts. Then the contract $\mathcal{C} \vee \mathcal{C}'$ is equal to the disjunction of the following three contracts:*

1. $(A \cup \neg (G - G'), G - G')$
2. $(A' \cup \neg (G' - G), G' - G)$
3. $(A \cap A' \cup \neg (G \cap G'), G \cap G')$

Proof. Obtained by dualizing the proof of Theorem 5.2.1. □

As with merging, composition is the third element of this factorization of the LUB. These two factorizations show that there is great symmetry between the notions of composition and merging. Both operations appear as a component of the factorizations of elementary operations of contracts, namely conjunction and disjunction, operations which are generated from the contract partial order. Figure 5.2 shows how merging, composition, conjunction, and disjunction are ordered.

5.3 Decomposition of contracts and separation of viewpoints

In the last chapter, we discussed the operation of quotient. In order to support a modular design process, the ability to *decompose* contracts into simpler ones is crucial; we called

(a) Composition and quotient

(b) Merging and separation

Figure 5.3: Let I_1 be a contract and define the functors $\mathcal{P}\,I = I \parallel I_1$, $\mathcal{M}\,I = I \bullet I_1$, $\mathcal{Q}\,I = I \,/\, I_1$, and $\mathcal{S}\,I = I \div I_1$. Then $\mathcal{P}$ is a left adjoint of $\mathcal{Q}$, and $\mathcal{M}$ is a right adjoint of $\mathcal{S}$.

quotient the operation that allows us to carry out decomposition. The quotient is defined in terms of composition and refinement. The output of the quotient, denoted $\mathcal{C} \,/\, \mathcal{C}_1$, has the property that its composition with $\mathcal{C}_1$ refines $\mathcal{C}$, and is maximal for this property in the refinement order:

$$\forall \mathcal{C}'. \quad \mathcal{C}' \parallel \mathcal{C}_1 \leq \mathcal{C} \iff \mathcal{C}' \leq \mathcal{C} \,/\, \mathcal{C}_1. \tag{5.1}$$

Suppose $\mathcal{C}$ corresponds to a top-level specification, and $\mathcal{C}_1$ to the specification of a component that will be used in the design. The quotient $\mathcal{C} \,/\, \mathcal{C}_1$ yields the specification of the functionality that $\mathcal{C}_1$ is missing for it to refine $\mathcal{C}$. Thus, the quotient is key in the decomposition of specifications.

We use the language of categories to describe some transformations between contracts. Categories are composed of objects and arrows between these objects. For instance, in the category of sets, objects are sets, and arrows are functions between sets. Functors are transformations between categories; they map objects to objects and arrows to arrows. Since refinement is a partial order for contracts, we can speak about a category of contracts, in which objects are contracts and an arrow from contract $\mathcal{C}$ to contract $\mathcal{C}'$ exists if and only if $\mathcal{C} \leq \mathcal{C}'$. We use the language of category theory to point out that some operations we have discussed have deep connections to each other (i.e., are not arbitrary definitions). For an in-depth treatment of category theory, see Mac Lane [138].

Let $\mathcal{C}_1$ be a contract. Let $\mathcal{P}(\mathcal{C}) = \mathcal{C} \parallel \mathcal{C}_1$ be an endofunctor (i.e., a transformation of objects within the same category) in the category of contracts. Let $\mathcal{Q}(\mathcal{C}) = \mathcal{C} \,/\, \mathcal{C}_1$ be another endofunctor. Then the definition (5.1) can be represented graphically as the universal property that the arrow α exists if and only if β exists in the diagram shown in Figure 5.3a. This means that $\mathcal{P}$ is a left adjoint to the functor $\mathcal{Q}$. From this universal property of the quotient, we can derive a closed-form expression which permits its calculation:

Theorem 5.3.1 (Theorem 4.2.5)**.** *Let $\mathcal{C}$ and $\mathcal{C}_1$ be contracts. Then $\mathcal{C}_q$ is the quotient between contracts $\mathcal{C}$ and $\mathcal{C}_1$, written $\mathcal{C}_q = \mathcal{C} \,/\, \mathcal{C}_1$, if and only if*

$$G_q = G \cap A_1 \cup \neg\,(A \cap G_1) \quad \textit{and} \quad A_q = A \cap G_1.$$

There exists a dual operation related to merging. We call *separation* the operation that allows us to separate a viewpoint from a given merged contract. Given a merged contract $\mathcal{C}$

and a viewpoint $\mathcal{C}_1$, the separation $\mathcal{C} \div \mathcal{C}_1$ is defined as follows:

$$\forall \mathcal{C}'. \quad \mathcal{C} \leq \mathcal{C}' \bullet \mathcal{C}_1 \iff \mathcal{C} \div \mathcal{C}_1 \leq \mathcal{C}'. \tag{5.2}$$

In the category of contracts, if we let $\mathcal{M}$ be the endofunctor $\mathcal{M}(\mathcal{C}) = \mathcal{C} \bullet \mathcal{C}_1$ and $\mathcal{S}(\mathcal{C}) = \mathcal{C} \div \mathcal{C}_1$, we observe that the given universal property (5.2) means that, in Figure 5.3b, arrow γ exists if and only if arrow δ exists. It follows that S is a left adjoint of the functor M. From the universal property of separation, we can obtain an explicit form that enables its computation:

Theorem 5.3.2 (Separation)**.** *Let $\mathcal{C}$ and $\mathcal{C}_1$ be contracts. Then $\mathcal{C}_s$ is the separation of contracts $\mathcal{C}$ and $\mathcal{C}_1$, written $\mathcal{C}_s = \mathcal{C} \div \mathcal{C}_1$, if and only if*

$$G_s = G \cap A_1 \quad \textit{and} \quad A_s = A \cap G_1 \cup \neg (G \cap A_1).$$

Proof. We wish to show that $\mathcal{C}_s$ satisfies

$$\forall \mathcal{C}'. \quad \mathcal{C} \leq \mathcal{C}' \bullet \mathcal{C}_1 \iff \mathcal{C}_s \leq \mathcal{C}'.$$

Suppose $\mathcal{C}_s \leq \mathcal{C}'$ for a contract $\mathcal{C}'$. Let $\mathcal{C}_2 = \mathcal{C}_1 \bullet \mathcal{C}_s$. Expanding, we have

$$\begin{aligned} A_2 &= A_1 \cap (G_1 \cap A \cup \neg G) \quad \text{and} \\ G_2 &= G \cup \neg A_1. \end{aligned}$$

Clearly, $A_2 \leq A$ and $G \leq G_2$, so $\mathcal{C} \leq \mathcal{C}_2$. Since merging is monotonic with respect to refinement, $\mathcal{C} \leq \mathcal{C}_2 \leq \mathcal{C}' \parallel \mathcal{C}_1$.

Conversely, suppose $\mathcal{C} \leq \mathcal{C}''$ for $\mathcal{C}'' = \mathcal{C}' \bullet \mathcal{C}_1$. We wish to show that $\mathcal{C}_s \leq \mathcal{C}'$. From the assumption $\mathcal{C} \leq \mathcal{C}''$, we observe that

$$\begin{aligned} G \leq G'' = G' \cap G_1 \cup \neg (A' \cap A_1) \leq G' \cup \neg A_1 \\ \therefore \quad G_s = G \cap A_1 \leq G'. \end{aligned} \tag{5.3}$$

Rewriting the left-hand side of (5.3), we have

$$\begin{aligned} \neg (G' \cap G_1) \cap (A' \cap A_1) \leq \neg G \\ \therefore \quad A' \leq \neg G \cup (G' \cap G_1) \cup \neg A_1 \leq \neg G \cup G_1. \end{aligned} \tag{5.4}$$

From the hypothesis $\mathcal{C} \leq \mathcal{C}''$, we have $A' \cap A_1 \leq A$. This constraint together with (5.4) gives us

$$A' \leq (\neg G \cup G_1) \cap (A \cup \neg A_1) = A \cap G_1 \cup \neg (G \cap A_1) = A_s.$$

This result and (5.3) imply that $\mathcal{C}_s \leq \mathcal{C}'$. □

Note the duality between quotient and separation. Suppose we are given a high level specification $\mathcal{C}$ of a design, and the specification $\mathcal{C}'$ of a component which will be used in the design. The quotient represents the most relaxed missing specification, $\mathcal{C}/\mathcal{C}'$, such that this missing specification in composition with $\mathcal{C}'$ refines $\mathcal{C}$. Merging, on the contrary, works as follows: suppose we are given a specification $\mathcal{C}$ and suppose we are told that a specification $\mathcal{C}'$ is part of a *covering* of $\mathcal{C}$ (i.e., a set of specifications whose merging *is refined* by $\mathcal{C}$); separation gives us the strictest specification which, when merged with $intf'$, is refined by $\mathcal{C}$. In other words, quotient is used to find decompositions, while separation is used to find coverings (e.g., abstractions) of specifications.

To illustrate quotient and separation, suppose we have a top level specification $\mathcal{C} = (A, G_1 \cap G_2 \cup \neg A)$. This top level specification requires environments to satisfy A, and components to satisfy $G_1 \cap G_2 \cup \neg A$. Suppose an existing component satisfies the contract $\mathcal{C}' = (A', G_1 \cup \neg A')$, where $A \subseteq A'$, i.e., this component provides part of the requirements of the top-level contract. We expect the quotient to tell that we need another component which implements the guarantees G_2. The quotient yields

$$\mathcal{C} \,/\, \mathcal{C}' = (A \cap G_1, G_2 \cup \neg\,(A \cap G_1)),$$

as we expected. Note that the quotient allows its implementations to make use of the guarantees G_1 as an assumption. In other words, the quotient allows its implementations to expect that the components satisfying the specification $\mathcal{C}'$ will do their job.

Now consider the following application of separation. Instead of looking for decompositions of a specification, we look for coverings of a specification. Suppose we have a top-level specification $\mathcal{C}' = (A, G \cup \neg A)$ that we wish to implement, say, through synthesis. Suppose the implementation resulting from synthesis has the specification $\mathcal{C} = (A \cap A_e, G' \cup \neg\,(A \cap A_e))$, where $G' \cup \neg\,(A \cap A_e) \subseteq G \cup \neg\,(A \cap A_e)$. That is, the implementation fails to be a refinement of the top-level specification because the implementation uses more assumptions. Note that this scenario is rather typical: top-level specifications often fail to include assumptions which are necessary for an implementation to work. These additional requirements are captured by A_e. We wish to compute the smallest specification we need to add to $\mathcal{C}'$ so that $\mathcal{C}'$ merged with this missing specification covers $\mathcal{C}$. Computing separation yields

$$\mathcal{C} \div \mathcal{C}' = (A_e \cup \neg\,(A)\,, G' \cap A \cup \neg\,(A_e \cup \neg A)).$$

We can abstract this separation result to the contract $(A_e, G' \cup \neg A_e)$ (verification that this is indeed an abstraction is left to the reader). This contract adds the missing requirements and enforces the stricter guarantees G'. If the previous guarantees were acceptable, one can abstract this contract even further to $(A_e, \mathcal{B}(\Sigma))$, where $\mathcal{B}(\Sigma)$ is the set of all behaviors under consideration.

The operations we have introduced have identities that characterize them in the contract lattice.

$\mathcal{C} \parallel \bot = \bot$ $\mathcal{C} \bullet \bot = (A, \neg(A))$	$\mathcal{C} \parallel \top = (\neg(G), G)$ $\mathcal{C} \bullet \top = \top$	$\mathcal{C} \parallel 1 = \mathcal{C}$ $\mathcal{C} \bullet 1 = \mathcal{C}$
$\mathcal{C} / \bot = \top$ $\bot / \mathcal{C} = (G, \neg(G))$	$\mathcal{C} / \top = (A, \neg(A))$ $\top / \mathcal{C} = \top$	$\mathcal{C} / 1 = \mathcal{C}$ $1 / \mathcal{C} = (G, A)$
$\mathcal{C} \div \bot = (\neg(G), G)$ $\bot \div \mathcal{C} = \bot$	$\mathcal{C} \div \top = \bot$ $\top \div \mathcal{C} = (\neg(A), A)$	$\mathcal{C} \div 1 = \mathcal{C}$ $1 \div \mathcal{C} = (G, A)$

Table 5.1: Behavior of composition, quotient, merging, and separation with respect to the distinguished elements of the theory of contracts

Definition 5.3.3 (Composition and merging identity)**.** *A composition identity, denoted* 1_c*, is a contract such that* $\mathcal{C} \parallel 1_c = 1_c \parallel \mathcal{C} = \mathcal{C}$*. Likewise, a merging identity, denoted* 1_m*, satisfies* $\mathcal{C} \bullet 1_m = 1_m \bullet \mathcal{C} = \mathcal{C}$*.*

The definition of the identities does not imply their uniqueness. The following lemma settles this issue:

Lemma 5.3.4. *Let* Σ *be the union of all alphabets over which components are defined. The contract identities just introduced and the contracts* $\bot$ *and* $\top$ *have the following explicit forms:* $\top = (\emptyset, \mathcal{B}(\Sigma))$*,* $\bot = (\mathcal{B}(\Sigma), \emptyset)$*, and* $1_m = 1_c = (\mathcal{B}(\Sigma), \mathcal{B}(\Sigma))$*. Since both identities are equal, we call* $1 = 1_c = 1_m$ *the identity.*

How do these distinguished elements behave with respect to the contract operations? These relations are shown in Table 5.1. We observe in this table that taking the quotient or separation from the identity results in a contract with flipped requirements and guarantees. This behavior motivates the following definition:

Definition 5.3.5 (Reciprocal)**.** *Let* $I = (A, G)$*. Its reciprocal, denoted* I^{-1}*, is given by* $I^{-1} = 1 / \mathcal{C} = 1 \div \mathcal{C} = (G, A)$*.*

Finally, let $\mathcal{C} = (A, G)$ be a contract defined over an alphabet Σ. Table 5.2 provides some identities pertaining merging and composition and their adjoint operations.

$\mathcal{C} \parallel \mathcal{C} = \mathcal{C}$	$\mathcal{C} / \mathcal{C} = (A \cap G, \mathcal{B}(\Sigma))$	$\mathcal{C} / \mathcal{C}' = \mathcal{C} \bullet \mathcal{C}'^{-1}$
$\mathcal{C} \bullet \mathcal{C} = \mathcal{C}$	$\mathcal{C} \div \mathcal{C} = (\mathcal{B}(\Sigma), A \cap G)$	$\mathcal{C} \div \mathcal{C}' = \mathcal{C} \parallel \mathcal{C}'^{-1}$

Table 5.2: Some properties of composition, merging, and their adjoints

The operation of reciprocal allows us to create a contract representing the perspective of the environment in which the design operates (as the reciprocal flips requirements and guarantees). We obtained several identities showing how the reciprocal interacts with the other

contract operations. It will be future work to better understand the role of the reciprocal in system design methodologies.

5.4 Multiviewpoint design

We can use the device of merging to analyze systems described under different viewpoints, and derive stronger combined results. To illustrate the procedure, we employ the case study introduced by Damm et al. [49] and there solved by manually combining a timing and a safety specification. We show that the application of our operators produces a more accurate result, which refines the one derived there.

The example consists of a redundant wheel brake system composed of a dual *Brake System Control Unit* (BSCU) and a Hydraulic actuator. We will be concerned mainly with the BSCU, which is shown schematically in Figure 5.4. The two units receive information regard-

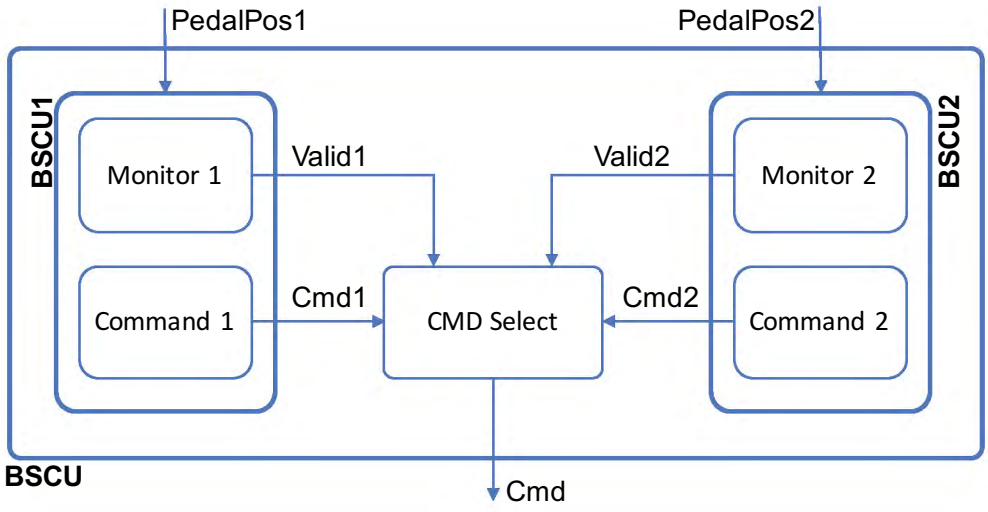


Figure 5.4: The Brake System Control Unit [49]

ing the position of the brake pedal, and deliver a brake command. The switch determines which of the two versions of the command to forward on the basis of the valid signals coming from the monitoring components. Without going into the details, the unit can sustain a single fault (at least one unit will have a valid signal), and must produce a command within 5 ms from a change in the pedal position. The specification makes use of the *Requirements Specification Language*, a natural-looking formal pattern-based assertion language. The timing and the safety analysis produce two contracts of the form $\mathcal{C} = ((H, R), (A, G))$, where H is the universe of behaviors (the requirements make no assumptions), R are the strong

assumptions, A are the weak assumptions and G are the guarantees[1]. The contracts for the above properties are:

Safety contract	
R:	*fail*(PedalPos1) **and** *fail*(PedalPos2) **do not occur**
A:	*No double fault*
G:	Valid1 **or** Valid2

Timing contract	
R:	PedalPos1 == PedalPos2
A:	Valid1 **or** Valid2
G:	**Delay from** *change*(PedalPos1) **or** *change*(PedalPos2) **to** *change*(Cmd) **within** [0, 5] ms

Merging the two contracts requires taking the intersection of the requirements after inverse projection, as well as the intersection of the guarantees, which we extend with the complement of the weak assumptions to obtain a canonical form. The addition of the complement of the requirements is not shown for brevity, as the semantics of the contract is not changed.

Merged contract	
R:	*fail*(PedalPos1) **and** *fail*(PedalPos2) **do not occur and** PedalPos1 == PedalPos2
A:	null
G:	((Valid1 **or** Valid2) **or** $\overline{\textit{No double fault}}$) **and** ((**Delay...**) **or** $\overline{(\text{Valid1 } \textbf{or } \text{Valid2})}$)

By logical manipulation, simplification and extracting the weak assumptions, we obtain:

Merged contract	
R:	*fail*(PedalPos1) **and** *fail*(PedalPos2) **do not occur and** PedalPos1 == PedalPos2
A:	(Valid1 **or** Valid2) **or** *No double fault*
G:	((Valid1 **or** Valid2) **and** (**Delay...**))

The result shows that under the assumption that the inputs are correct, and if there is no double fault, the system guarantees the stated delay between the change of the pedal position and the braking command. While the results in the original paper are only informally stated, they do appear to lack the first term of the weak assumption, making ours a more refined contract (it accepts more environments). The first term is necessary, as the original specification only expressed the forward implication (no double fault implies at least one valid signal is asserted), but not the converse. Having a coherent theory is therefore fundamental

[1] In the original paper [49], the strong assumptions were denoted by A, the weak assumptions by B and the guarantees by G.

to ensure correctness, and to cover all corner cases. In this case, the risk is to have contract satisfaction despite a possible double failure. This may or may not be important in the context of the system, but we believe the designer should be aware of the possibility and take action as required.

Separation can be used to go back to the individual contracts. For instance, if we separate the timing viewpoint and project away its variables we reconstruct the safety contract:

Separating timing viewpoint from merged contract	
R:	*fail*(PedalPos1) **and** *fail*(PedalPos2) **do not occur**
A:	*No double fault* **or** (Valid1 **or** Valid2)
G:	(Valid1 **or** Valid2)

Again, the result is a refinement of the original contract, recovering the implicit single implication.

Chapter 6

The algebra of assume-guarantee contracts

This chapter summarizes the algebra of assume-guarantee contracts. It defines AG contracts, provides all their known operations, and shows all known ways in which these operations are related. The chapter has the character of a fast tutorial on AG contracts. In addition, the chapter studies monoid and semiring structures for a contract algebra, and the mappings between such structures. Some of these mappings are used to define an action of a Boolean algebra on its contract algebra.

6.1 Introduction

AG contracts can be understood as specifications split in two pieces: (i) assumptions made on the environment, and (ii) responsibilities assigned to the object adhering to the specification when it is instantiated in an environment which meets the assumptions of the contract. Contracts were introduced to streamline the integration of complex systems and to support concurrent design. System integration pertains the composition of multiple design objects into a coherent whole. Suppose a company wishes to implement a system with a specification $\mathcal{C}$; they may realize that there are two sub-specifications $\mathcal{C}_1$ and $\mathcal{C}_2$ such that the composition of their implementations always yields an implementation for the top level spec. In the language of AG contracts, we would say that the composition of $\mathcal{C}_1$ and $\mathcal{C}_2$, written $\mathcal{C}_1 \parallel \mathcal{C}_2$, *refines* $\mathcal{C}$. This company may now develop an implementation M_1 for $\mathcal{C}_1$, and provide $\mathcal{C}_2$ to a third-party OEM in order for them to develop an implementation M_2. If M_2 is an implementation for $\mathcal{C}_2$, the original company knows that M_1 and M_2 can be composed and that this composition meets the top-level specification $\mathcal{C}$.

In this setting, frictions in the supply chain are alleviated as companies exchange formal specifications expressed as contracts. It is also a common design task to find missing components. Suppose our company again wants to implement a system with a specification $\mathcal{C}$ and suppose they know that to implement this design they will use a component M_1 with

specification $\mathcal{C}_1$. Contracts provide an operation called quotient which yields the specification whose implementations are *exactly* those components M' such that M_1 composed with M' meets the spec $\mathcal{C}$. The operation of quotient has uses in synthesis (when we have made incremental progress towards meeting a goal) and in every situation where we need to find missing components.

To say that contracts support concurrent design refers to another aspect of the design process. The design of some components involves multiple engineers working on different aspects of the same object. For example, a team may work on the functionality aspects of an IC, while another works on its timing characterization. If the functionality team generates a spec $\mathcal{C}_f$, and the timing team generates a spec $\mathcal{C}_t$, the two teams can combine their specs into a single contract object $\mathcal{C}$ through an operation called *merging*. In the contract theory, these various aspects of a component are called *viewpoints*.

This chapter defines AG contracts and summarizes their known operations.

6.2 AG Contracts

To define AG contracts, our starting point is the notion of properties. Thus, before talking about contracts, we assume we have chosen a formalism to model our components and the metrics we wish to verify in our system (properties).

Definition 6.2.1. *A contract $\mathcal{C}$ is a pair of properties $\mathcal{C} = (A, G)$. We call A assumptions, and G guarantees.*

We now define what it means for a component to be an environment and an implementation for a contract.

Definition 6.2.2. *Let $\mathcal{C} = (A, G)$ be a contract. We say that a component E is an environment for $\mathcal{C}$, written $E \models^E \mathcal{C}$, if $E \models A$.*

Environments are those components which meet the assumptions of a contract. Implementations are those which meet the guarantees of the contract when operating in an environment accepted by the contract:

Definition 6.2.3. *Let $\mathcal{C} = (A, G)$ be a contract. We say that a component M is an implementation for $\mathcal{C}$, written $M \models^M \mathcal{C}$, if $M \parallel E \models G$ for every environment E of $\mathcal{C}$.*

Now that we have definitions for environments and implementations, we define a relation on contracts that declares two contracts equivalent when they have the same environments and the same implementations:

Definition 6.2.4. *Let $\mathcal{C}$ and $\mathcal{C}'$ be two contracts. We say they are equivalent when*

$$E \models^E \mathcal{C} \text{ if and only if } E \models^E \mathcal{C}' \text{ and}$$
$$M \models^M \mathcal{C} \text{ if and only if } M \models^M \mathcal{C}'.$$

This means that for $\mathcal{C} = (A, G)$ and $\mathcal{C}' = (A', G')$ to be equivalent, we must have $A = A'$ and $G \cap A = G' \cap A' = G' \cap A$ (because $A' = A$). The largest G' meeting this condition is $G' = G \cup \neg A$ (the complement is taken with respect to $\mathcal{B}$). Enforcing this constraint for a contract allows us to have a unique mathematical object for each set of environments and implementations. We thus define an AG contract in canonical form as follows:

Definition 6.2.5. *A contract in canonical form is a contract* $\mathcal{C} = (A, G)$ *satisfying* $A \cup G = \mathcal{B}$.

From now on, we assume all contracts are in canonical form.

6.3 Order

Suppose $\mathcal{C}$ and $\mathcal{C}'$ are two contracts. We say that $\mathcal{C}$ is a refinement of $\mathcal{C}'$, written $\mathcal{C} \leq \mathcal{C}'$, when all implementations of $\mathcal{C}$ are implementations of $\mathcal{C}'$ and all environments of $\mathcal{C}'$ are environments of $\mathcal{C}$, i.e.,

$$M \models^M \mathcal{C} \Rightarrow M \models^M \mathcal{C}' \text{ and}$$
$$E \models^E \mathcal{C}' \Rightarrow E \models^E \mathcal{C}.$$

The association we make of a specification being a refinement is that it is harder to meet than another. This is why we say that a specification accepting more environments is a refinement of one accepting less. We can express this order relation using assumptions and guarantees.

Let $\mathcal{C} = (A, G)$ and $\mathcal{C}' = (A', G')$ be two contracts. Then $\mathcal{C} \leq \mathcal{C}'$ when

$$G \subseteq G' \text{ and } A' \subseteq A.$$

6.4 Duality

There is a unary operation which is quite helpful in revealing structure for AG contracts. Let $\mathcal{C} = (A, G)$ be a contract. We define a unary operation called reciprocal as follows:

$$C^{-1} = (G, A).$$

This operation flips environments and implementations, i.e., it gives us the "environment view" of the specification $\mathcal{C}$. Note that the reciprocal is a well-defined operation on AG contracts because $A \cup \neg G = \neg(\neg A \cap G) = \neg\neg A = A$.

Let $\circ$ and $\star$ be two binary operations on AG contracts, we say that the operations are dual when

$$\mathcal{C} \circ \mathcal{C}' = \left(\mathcal{C}^{-1} \star \mathcal{C}'^{-1}\right)^{-1}.$$

6.5 Conjunction and disjunction

The notion of order provides a lattice structure to AG contracts. Given contracts $\mathcal{C} = (A, G)$ and $\mathcal{C}' = (A', G')$, their meet (GLB) and join (LUB) are given by

$$\mathcal{C} \wedge \mathcal{C}' = (A \cup A', G \cap G') \text{ and}$$
$$\mathcal{C} \vee \mathcal{C}' = (A \cap A', G \cup G').$$

We leave it to the reader to verify that conjunction and disjunction are monotonic with respect to the refinement order. Also, we observe that conjunction is the dual of disjunction:

$$\mathcal{C} \wedge \mathcal{C}' = (G \cap G', A \cup A')^{-1} = ((G, A) \vee (G', A'))^{-1} = \left(\mathcal{C}^{-1} \vee \mathcal{C}'^{-1}\right)^{-1}.$$

6.6 Composition

The notion of composition of AG contracts yields the specification of systems obtained from composing implementations of each of the contracts being composed. This operation is defined by axiom as follows:

Suppose $\mathcal{C}_1$ and $\mathcal{C}_2$ are two specifications to be composed. Call $\mathcal{C}$ the composite specification. Let M_1 and M_2 be arbitrary implementations of $\mathcal{C}_1$ and $\mathcal{C}_2$, respectively, and let E be any environment of $\mathcal{C}$. We define $\mathcal{C}$ to be the smallest contract satisfying the following constraints:

- The composite $M_1 \parallel M_2$ is an implementation of $\mathcal{C}$;
- the composite $M_1 \parallel E$ is an environment of $\mathcal{C}_2$; and
- the composite $M_2 \parallel E$ is an environment of $\mathcal{C}_1$.

The first requirement states that composing implementations of the specs being composed yields an implementation of the composite spec. The second requirement states that instantiating an implementation of $\mathcal{C}_1$ in an environment of the composite spec yields an environment for $\mathcal{C}_2$. And the last requirement is the analogous statement for $\mathcal{C}_1$. This principle, which states how to compose specifications split between environment and implementation requirements, was stated for the first time by M. Abadi and L. Lamport [1].

We can obtain a closed-form expression of this principle for AG contracts:

Proposition 6.6.1. *Let $\mathcal{C}_1 = (A_1, G_1)$ and $\mathcal{C}_2 = (A_2, G_2)$ be two AG contracts. Their composition, denoted $\mathcal{C}_1 \parallel \mathcal{C}_2$, is given by*

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

Proof. The following statement instantiates the composition principle using the symbols we just defined:

$$\mathcal{C}_1 \parallel \mathcal{C}_2 = \bigwedge \left\{ \mathcal{C} \;\middle|\; \begin{bmatrix} M_1 \parallel M_2 \models^M \mathcal{C} \\ M_1 \parallel E \models^E \mathcal{C}_2 \\ M_2 \parallel E \models^E \mathcal{C}_1 \end{bmatrix} \text{ for all } \begin{bmatrix} M_1 \models^M \mathcal{C}_1 \\ M_2 \models^M \mathcal{C}_2 \\ E \models^E \mathcal{C} \end{bmatrix} \right\}.$$

Now we plug-in definitions:

$$\begin{aligned} \mathcal{C}_1 \parallel \mathcal{C}_2 &= \bigwedge \left\{ \mathcal{C} \;\middle|\; \begin{bmatrix} M_1 \parallel M_2 \models^M \mathcal{C} \\ M_1 \parallel E \subseteq A_2 \\ M_2 \parallel E \subseteq A_1 \end{bmatrix} \text{ for all } \begin{bmatrix} M_1 \subseteq G_1 \\ M_2 \subseteq G_2 \\ E \models^E \mathcal{C} \end{bmatrix} \right\} \\ &= \bigwedge \left\{ \mathcal{C} \;\middle|\; \begin{bmatrix} G_1 \cap G_2 \models^M \mathcal{C} \\ E \subseteq (A_2 \cup \neg G_1) \cap (A_1 \cup \neg G_2) \end{bmatrix} \text{ for all } E \models^E \mathcal{C} \right\}. \end{aligned}$$

Since a contract (A, G) satisfies $G \supseteq \neg A$, it also satisfies $A \supseteq \neg G$, which means that the upper bound on E can be written $(A_2 \cup \neg G_1) \cap (A_1 \cup \neg G_2) = A_1 \cap A_2 \cup \neg(G_1 \cap G_2)$. Thus,

$$\mathcal{C}_1 \parallel \mathcal{C}_2 = \bigwedge \left\{ \mathcal{C} \;\middle|\; \begin{bmatrix} G_1 \cap G_2 \models^M \mathcal{C} \\ E \subseteq A_1 \cap A_2 \cup \neg(G_1 \cap G_2) \end{bmatrix} \text{ for all } E \models^E \mathcal{C} \right\}.$$

The expression imposes a lower-bounded to the guarantees of $\mathcal{C}$ and an upper bound to the assumptions. The GLB requires us to find $\mathcal{C}$ with the smallest guarantees and largest assumptions. The contract simultaneously satisfying those constraints is

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

But we observe that this contract satisfies the constraint of a contract. Thus, the closed-form expression stated in the proposition yields the composition according to the principle. □

We state without proof an important property of composition:

Proposition 6.6.2. *Composition of AG contracts is monotonic with respect to the refinement order.*

6.7 Strong merging (or merging)

We said that AG contracts are used not only to handle the specifications of multiple components comprising a system, but also the various viewpoints of the same design element. Suppose $\mathcal{C}_1$ and $\mathcal{C}_2$ are specifications corresponding to different aspects to the same design object, e.g., functionality and power. We define their merger, denoted $\mathcal{C}_1 \bullet \mathcal{C}_2$, to be the

contract which guarantees the guarantees of both specifications when the assumptions of both specifications are respected, that is,

$$\mathcal{C}_1 \bullet \mathcal{C}_2 = (A_1 \cap A_2, G_1 \cap G_2 \cup \neg(A_1 \cap A_2)) .$$

Observe that this contract is equivalent to the contract $(A_1 \cap A_2, G_1 \cap G_2)$, which is exactly what we defined merging to be.

Moreover, merging and composition are duals.

6.8 Adjoints

We have introduced four operations on AG contracts: two were obtained from the partial order, and two by axiom. Now we obtain the adjoints of these operations.

6.8.1 Quotient (or residual)

Let $\mathcal{C}$ and $\mathcal{C}'$ be two AG contracts. The quotient[1], denoted $\mathcal{C}/\mathcal{C}'$, is defined as the largest AG contract $\mathcal{C}''$ satisfying

$$\mathcal{C}' \parallel \mathcal{C}'' \leq \mathcal{C}.$$

Due to the fact that the quotient is the largest contract with this property, Proposition 6.6.2 tells us that any of its refinements also has this property.

If we interpret $\mathcal{C}$ as a top-level spec that our system has to meet (e.g., the spec of a vehicle), and $\mathcal{C}'$ as the specification of a subset of the design for which we already have an implementation (e.g., a powertrain), then the quotient is the specification whose implementations are exactly those components which, if added to our partial design, would yield a system meeting the top-level specification. The following proposition gives us a closed-form expression for the quotient of AG contracts:

Proposition 6.8.1 (Theorem 4.2.5)**.** *Let $\mathcal{C} = (A, G)$ and $\mathcal{C}' = (A', G')$ be two AG contracts. The quotient, denoted $\mathcal{C}/\mathcal{C}'$, is given by*

$$\mathcal{C}/\mathcal{C}' = (A \cap G', G \cap A' \cup \neg(A \cap G')) .$$

For examples of the use of the quotient to identify missing components, see Sections 4.3.3 and 4.5. For an in-depth study of the notion of a quotient across several compositional theories, see Chapter 4.

We can readily show that

$$\mathcal{C}/\mathcal{C}' = \mathcal{C} \bullet (\mathcal{C}')^{-1}. \tag{6.1}$$

[1] Also called residual in the literature.

6.8.2 Separation

Just like composition has an adjoint operation (the quotient), merging has an adjoint. For contracts $\mathcal{C}$ and $\mathcal{C}'$, we define the operation of *separation*, denoted $\mathcal{C} \div \mathcal{C}'$, as the smallest contract $\mathcal{C}''$ satisfying

$$\mathcal{C} \leq \mathcal{C}' \bullet \mathcal{C}''.$$

This operation has a closed-form solution:

Proposition 6.8.2 (Theorem 5.3.2)**.** *Let $\mathcal{C} = (A, G)$ and $\mathcal{C}' = (A', G')$ be two AG contracts. Then*

$$\mathcal{C} \div \mathcal{C}' = (A \cap G' \cup \neg(G \cap A'), G \cap A').$$

Separation obeys the identity:

$$\mathcal{C} \div \mathcal{C}' = \mathcal{C} \parallel (\mathcal{C}')^{-1}.$$

From this identity and (6.1), it follows that quotient and separation are duals. For examples of merging and separation, see Chapter 5.

6.8.3 Implication and coimplication

Given contracts $\mathcal{C}$ and $\mathcal{C}'$, the definition of implication, denoted $\mathcal{C}' \to \mathcal{C}$, in a lattice is as follows:

$$\forall \mathcal{C}''.\ \mathcal{C}'' \wedge \mathcal{C}' \leq \mathcal{C} \Leftrightarrow \mathcal{C}'' \leq (\mathcal{C}' \to \mathcal{C}).$$

In other words, $\mathcal{C}' \to \mathcal{C}$ is the largest contract $\mathcal{C}''$ satisfying $\mathcal{C}'' \wedge \mathcal{C}' \leq \mathcal{C}$. The following proposition tells us how to compute this object:

Proposition 6.8.3. *Let $\mathcal{C} = (A, G)$ and $\mathcal{C}' = (A', G')$ be two contracts. Implication has the closed form expression*

$$\mathcal{C}' \to \mathcal{C} = ((A \cap \neg A') \cup (G' \cap \neg G), G \cup \neg G').$$

Proof. Let $\mathcal{C}_i$ be the contract stated in the proposition. Observe that $\mathcal{C}_i \wedge \mathcal{C}' = \mathcal{C} \wedge \mathcal{C}'$. Thus, by the monotonicity of conjunction, if $\mathcal{C}'' \leq \mathcal{C}_i$, then $\mathcal{C}'' \wedge \mathcal{C}' \leq \mathcal{C}$.

Now write $\mathcal{C}''$ as $\mathcal{C}'' = (A'', G'')$ and assume that $\mathcal{C}'' \wedge \mathcal{C}' \leq \mathcal{C}$. Then

$$\begin{aligned} &G'' \cap G' \leq G \text{ and} \\ &A'' \cup A' \geq A. \end{aligned}$$

From this we conclude that

$$G'' \leq G \cup \neg G' \text{ and} \tag{6.2}$$
$$A'' \geq A \cap \neg A'. \tag{6.3}$$

From (6.2) and the fact that $A'' \geq \neg G''$ (which follows from the definition of AG contracts), we obtain $A'' \geq G' \cap \neg G$. This result and (6.3) yield

$$A'' \geq (A \cap \neg A') \cup (G' \cap \neg G).$$

This expression and (6.2) mean that $\mathcal{C}'' \leq \mathcal{C}_i$, completing the proof. □

Dually, we can ask what is the smallest contract $\mathcal{C}''$ satisfying

$$\mathcal{C}'' \vee \mathcal{C}' \geq \mathcal{C}.$$

We will denote this object $\mathcal{C}' \nrightarrow \mathcal{C}$. A similar proof yields the following proposition.

Proposition 6.8.4. *Let $\mathcal{C} = (A, G)$ and $\mathcal{C}' = (A', G')$ be two contracts. The smallest contract $\mathcal{C}''$ satisfying $\mathcal{C}'' \vee \mathcal{C}' \geq \mathcal{C}$ has the closed form expression*

$$\mathcal{C}' \nrightarrow \mathcal{C} = (A \cup \neg A', (G \cap \neg G') \cup (A' \cap \neg A)).$$

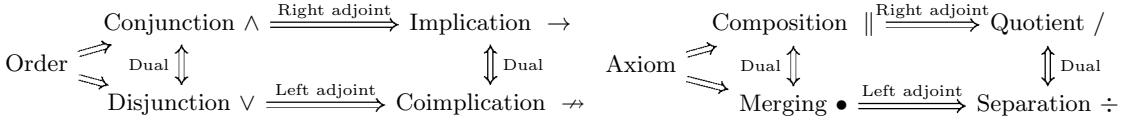
We observe that

$$\mathcal{C}' \to \mathcal{C} = (G \cup \neg G', (A \cap \neg A') \cup (G' \cap \neg G))^{-1} = \left((\mathcal{C}')^{-1} \nrightarrow \mathcal{C}^{-1}\right)^{-1},$$

which shows that implication and coimplication are duals.

6.9 Summary of binary operations

The following diagram shows how all AG contract operations are related.



Tables 6.1 and 6.2 show the closed-form expressions and the duality relations for all operations.

Composition and merging	$\mathcal{C}_1 \parallel \mathcal{C}_2 = \left(\mathcal{C}_1^{-1} \bullet \mathcal{C}_2^{-1}\right)^{-1}$	$\mathcal{C}_1 \bullet \mathcal{C}_2 = \left(\mathcal{C}_1^{-1} \parallel \mathcal{C}_2^{-1}\right)^{-1}$
Quotient and separation	$\mathcal{C}/\mathcal{C}' = (\mathcal{C}^{-1} \div (\mathcal{C}')^{-1})^{-1}$	$\mathcal{C} \div \mathcal{C}' = (\mathcal{C}^{-1}/(\mathcal{C}')^{-1})^{-1}$
Conjunction and disjunction	$\mathcal{C}_1 \wedge \mathcal{C}_2 = \left(\mathcal{C}_1^{-1} \vee \mathcal{C}_2^{-1}\right)^{-1}$	$\mathcal{C}_1 \vee \mathcal{C}_2 = \left(\mathcal{C}_1^{-1} \wedge \mathcal{C}_2^{-1}\right)^{-1}$
Implication an coimplication	$\mathcal{C}' \to \mathcal{C} = ((\mathcal{C}')^{-1} \nrightarrow \mathcal{C}^{-1})^{-1}$	$\mathcal{C}' \nrightarrow \mathcal{C} = ((\mathcal{C}')^{-1} \to \mathcal{C}^{-1})^{-1}$

Table 6.1: Duality relations

Moreover, we have the following relations between composition, quotient, merging, and separation: $\mathcal{C}/\mathcal{C}' = \mathcal{C} \bullet (\mathcal{C}')^{-1}$ $\mathcal{C} \div \mathcal{C}' = \mathcal{C} \parallel (\mathcal{C}')^{-1}$

Conjunction	Disjunction
$\mathcal{C} \wedge \mathcal{C}' = (A \cup A', G \cap G')$	$\mathcal{C} \vee \mathcal{C}' = (A \cap A', G \cup G')$
Composition	Merging
$\mathcal{C}_1 \parallel \mathcal{C}_2 = (A_1 \cap A_2 \cup \neg(G_1 \cap G_2), G_1 \cap G_2)$	$\mathcal{C}_1 \bullet \mathcal{C}_2 = (A_1 \cap A_2, G_1 \cap G_2 \cup \neg(A_1 \cap A_2))$
Quotient	Separation
$\mathcal{C}/\mathcal{C}' = (A \cap G', G \cap A' \cup \neg(A \cap G'))$	$\mathcal{C} \div \mathcal{C}' = (A \cap G' \cup \neg(G \cap A'), G \cap A')$
Implication	Coimplication
$\mathcal{C}' \to \mathcal{C} = ((A \cap \neg A') \cup (G' \cap \neg G), G \cup \neg G')$	$\mathcal{C}' \nrightarrow \mathcal{C} = (A \cup \neg A', (G \cap \neg G') \cup (A' \cap \neg A))$

Table 6.2: Closed-form expressions of contract operations

Conjunction	Disjunction
$\mathcal{C} \wedge \mathcal{C}' = (a \vee a', g \wedge g')$	$\mathcal{C} \vee \mathcal{C}' = (a \wedge a', g \vee g')$
Composition	Merging
$\mathcal{C}_1 \parallel \mathcal{C}_2 = (a_1 \wedge a_2 \vee \neg(g_1 \wedge g_2), g_1 \wedge g_2)$	$\mathcal{C}_1 \bullet \mathcal{C}_2 = (a_1 \wedge a_2, g_1 \wedge g_2 \vee \neg(a_1 \wedge a_2))$
Quotient	Separation
$\mathcal{C}/\mathcal{C}' = (a \wedge g', g \wedge a' \vee \neg(a \wedge g'))$	$\mathcal{C} \div \mathcal{C}' = (a \wedge g' \vee \neg(g \wedge a'), g \wedge a')$
Implication	Coimplication
$\mathcal{C}' \to \mathcal{C} = ((a \wedge \neg a') \vee (g' \wedge \neg g), g \vee \neg g')$	$\mathcal{C}' \nrightarrow \mathcal{C} = (a \vee \neg a', (g \wedge \neg g') \vee (a' \wedge \neg a))$

Table 6.3: Closed-form expressions of operations for contracts over a Boolean algebra

6.10 Algebraic structures within contracts

Inspection of the various binary formulas for AG contracts (Table 6.2) suggests that contracts can be defined over any Boolean algebra, not just that corresponding to properties over a set of behaviors. Thus, for any Boolean algebra B, we have an associated contract algebra $\mathcal{C}(B)$ whose elements are all pairs $(a, b) \in B^2$ such that $a \vee b = 1$. The notions of order and the binary operations work exactly the same as for AG contracts over sets of behaviors. Table 6.3 summarizes these operations.

Let 0_B and 1_B be the bottom and top elements, respectively, of the Boolean algebra B. The contract $1 = (0_B, 1_B)$ is larger than any contract. $0 = (1_B, 0_B)$ is smaller than any contract. The contract $\mathrm{id} = (1_B, 1_B)$ is an identity for composition and merging. 1 is an identity for conjunction, and 0 for disjunction. Table 6.4 shows how various operations behave with respect to the distinguished elements.

6.10.1 Monoids

We recall that a monoid is a semigroup with identity, i.e., a set together with an associative binary operation and an identity element for that operation. A contract algebra contains several monoids:

	0	1	id
Conjunction	$\mathcal{C} \wedge 0 = 0$	$\mathcal{C} \wedge 1 = \mathcal{C}$	$(a,g) \wedge \mathrm{id} = (1_B, g)$
Disjunction	$\mathcal{C} \vee 0 = \mathcal{C}$	$\mathcal{C} \vee 1 = 1$	$(a,g) \vee \mathrm{id} = (a, 1_B)$
Composition	$\mathcal{C} \parallel 0 = 0$	$(a,g) \parallel 1 = (\neg g, g)$	$\mathcal{C} \parallel \mathrm{id} = \mathcal{C}$
Merging	$(a,g) \bullet 0 = (a, \neg a)$	$\mathcal{C} \bullet 1 = 1$	$\mathcal{C} \bullet \mathrm{id} = \mathcal{C}$
Quotient	$\mathcal{C}/0 = 1$ $0/(a,g) = (g, \neg g)$	$(a,g)/1 = (a, \neg a)$ $1/\mathcal{C} = 1$	$\mathcal{C}/\mathrm{id} = \mathcal{C}$ $\mathrm{id}/\mathcal{C} = \mathcal{C}^{-1}$
Separation	$(a,g) \div 0 = (\neg g, g)$ $0 \div \mathcal{C} = 0$	$\mathcal{C} \div 1 = 0$ $1 \div (a,g) = (\neg a, a)$	$\mathcal{C} \div \mathrm{id} = \mathcal{C}$ $\mathrm{id} \div \mathcal{C} = \mathcal{C}^{-1}$
Implication	$(a,g) \to 0 = (g, \neg g)$ $0 \to \mathcal{C} = 1$	$\mathcal{C} \to 1 = 1$ $1 \to \mathcal{C} = \mathcal{C}$	$(a,g) \to \mathrm{id} = (\neg a, 1_B)$ $\mathrm{id} \to (a,g) = (\neg g, g)$
Coimplication	$\mathcal{C} \nrightarrow 0 = 0$ $0 \nrightarrow \mathcal{C} = \mathcal{C}$	$(a,g) \nrightarrow 1 = (\neg a, a)$ $1 \nrightarrow \mathcal{C} = 0$	$(a,g) \nrightarrow \mathrm{id} = (1_B, \neg g)$ $\mathrm{id} \nrightarrow (a,g) = (a, \neg a)$

Table 6.4: Contract operations and the distinguished elements

Proposition 6.10.1. *$\mathcal{C}^M_\wedge(B) = (\mathcal{C}(B), \wedge, 1_B)$, $\mathcal{C}^M_\vee(B) = (\mathcal{C}(B), \vee, 0_B)$, $\mathcal{C}^M_\parallel(B) = (\mathcal{C}(B), \parallel, id)$, and $\mathcal{C}^M_\bullet(B) = (\mathcal{C}(B), \bullet, id)$ are idempotent, commutative monoids.*

Proof. We already know that 1 and 0 are, respectively, the identity elements of conjunction and disjunction. id is the identity for composition and merging. The idempotence of these operations follows immediately from their definitions. It remains to show that these operations are associative.

Let $\mathcal{C} = (a,g)$, $\mathcal{C}' = (a',g')$, and $\mathcal{C}'' = (a'',g'')$ be contracts.

- Conjunction.

$$\begin{aligned}\mathcal{C} \wedge (\mathcal{C}' \wedge \mathcal{C}'') &= (a,g) \wedge (a' \vee a'', g' \wedge g'') \\ &= (a \vee (a' \vee a''), g \wedge (g' \wedge g'')) = ((a \vee a') \vee a'', (g \wedge g') \wedge g'') \\ &= (a \vee a', g \wedge g') \wedge \mathcal{C}'' = (\mathcal{C} \wedge \mathcal{C}') \wedge \mathcal{C}''\end{aligned}$$

- Disjunction.

$$\begin{aligned}\mathcal{C} \vee (\mathcal{C}' \vee \mathcal{C}'') &= \left(\mathcal{C}^{-1} \wedge ((\mathcal{C}')^{-1} \wedge (\mathcal{C}'')^{-1})\right)^{-1} = \left((\mathcal{C}^{-1} \wedge (\mathcal{C}')^{-1}) \wedge (\mathcal{C}'')^{-1}\right)^{-1} \\ &= (\mathcal{C} \vee \mathcal{C}') \vee \mathcal{C}''\end{aligned}$$

- Composition.

$$\begin{aligned}\mathcal{C} \parallel (\mathcal{C}' \parallel \mathcal{C}'') &= (a, g) \parallel (\neg g' \vee \neg g'' \vee (a' \wedge a''), g' \wedge g'') \\ &= (\neg g \vee \neg g' \vee \neg g'' \vee (a \wedge a' \wedge a''), g \wedge (g' \wedge g'')) \\ &= (\neg g \vee \neg g' \vee \neg g'' \vee ((a \wedge a') \wedge a''), (g \wedge g') \wedge g'') \\ &= (\neg g \vee \neg g' \vee (a \wedge a'), g \wedge g') \parallel \mathcal{C}'' = (\mathcal{C} \parallel \mathcal{C}') \parallel \mathcal{C}''\end{aligned}$$

- Merging.

$$\begin{aligned}\mathcal{C} \bullet (\mathcal{C}' \bullet \mathcal{C}'') &= \left(\mathcal{C}^{-1} \parallel ((\mathcal{C}')^{-1} \parallel (\mathcal{C}'')^{-1})\right)^{-1} = \left((\mathcal{C}^{-1} \parallel (\mathcal{C}')^{-1}) \parallel (\mathcal{C}'')^{-1}\right)^{-1} \\ &= (\mathcal{C} \bullet \mathcal{C}') \bullet \mathcal{C}''\end{aligned}$$

□

It turns out these monoids are isomorphic:

Proposition 6.10.2. *The monoids $\mathcal{C}^M_\wedge(B)$, $\mathcal{C}^M_\vee(B)$, $\mathcal{C}^M_\parallel(B)$, and $\mathcal{C}^M_\bullet(B)$ are isomorphic.*

Proof. Due to the duality relations between conjunction and disjunction and between composition and merging, the reciprocal map provides monoid isomorphisms between $(\mathcal{C}(B), \wedge, 1)$ and $(\mathcal{C}(B), \vee, 0)$ and between $(\mathcal{C}(B), \parallel, \mathrm{id})$ and $(\mathcal{C}(B), \bullet, \mathrm{id})$.

We now show that the map $\theta_g : \mathcal{C}_\parallel(B) \to \mathcal{C}_\wedge(B)$ defined as

$$\theta_g : (a, g) \mapsto (\neg(a \wedge g), g)$$

is a monoid isomorphism.

Observe that $\theta_g^2(a, g) = \theta_g(\neg(a \wedge g), g) = (\neg(\neg(a \wedge g) \wedge g), g) = (a, g)$, so θ_g is an involution. Now we show it is a monoid homomorphism. Let $\mathcal{C} = (a, g)$ and $\mathcal{C}' = (a', g')$.

- $\theta_g(\mathrm{id}) = \theta_g(1, 1) = (0, 1) = 1$
- We verify whether θ_g commutes with the multiplications:

$$\begin{aligned}\theta_g(\mathcal{C} \parallel \mathcal{C}') &= \theta_g(\neg(g \wedge g') \vee (a \wedge a'), g \wedge g') \\ &= (\neg(a \wedge g \wedge a' \wedge g'), g \wedge g') = (\neg(a \wedge g) \vee \neg(a' \wedge g'), g \wedge g') \\ &= (\neg(a \wedge g), g) \wedge (\neg(a' \wedge g'), g') = \theta_g(\mathcal{C}) \wedge \theta_g(\mathcal{C}').\end{aligned}$$

As θ_g is an involution, we have to check that it is a monoid map from $\mathcal{C}_\wedge(B)$ to $\mathcal{C}_\parallel(B)$.

$$\begin{aligned}\theta_g(\mathcal{C} \wedge \mathcal{C}') &= \theta_g(a \vee a', g \wedge g') \\ &= (\neg(g \wedge g' \wedge (a \vee a')), g \wedge g') = (\neg(g \wedge g') \vee (\neg a \wedge \neg a'), g \wedge g') \\ &= \left(\neg(g \wedge g') \vee (\neg(a \wedge g) \wedge \neg(a' \wedge g'))\right, g \wedge g') = (\neg(a \wedge g), g) \parallel (\neg(a' \wedge g'), g') \\ &= \theta_g(\mathcal{C}) \parallel \theta_g(\mathcal{C}')\end{aligned}$$

□

The following diagram summarizes the isomorphisms of the four contract monoids:

$$\begin{array}{ccc} \mathcal{C}_{\wedge}^{M}(B) & \underset{\simeq}{\overset{(\cdot)^{-1}}{\longleftrightarrow}} & \mathcal{C}_{\vee}^{M}(B) \\ {\scriptstyle \theta_g}\uparrow\downarrow{\scriptstyle \wr} & & {\scriptstyle \wr}\uparrow\downarrow{\scriptstyle \theta_a} \\ \mathcal{C}_{\|}^{M}(B) & \underset{(\cdot)^{-1}}{\overset{\simeq}{\longleftrightarrow}} & \mathcal{C}_{\bullet}^{M}(B) \end{array}$$

The isomorphism θ_a is defined using the diagram, i.e.,

$$\theta_a(a,g) = \left(\theta_g(a,g)^{-1}\right)^{-1} = (\theta_g(g,a))^{-1} = (\neg(a \wedge g), a)^{-1} = (a, \neg(a \wedge g)).$$

Now suppose we have two Boolean algebras, B and B'. We will study the structure of the maps between the contract monoids associated with each Boolean algebra. Due to Proposition 6.10.2, it is sufficient to study the structure of the maps between the contract monoids $\mathcal{C}_{\|}^{M}(B)$ and $\mathcal{C}_{\|}^{M}(B')$. First we study maps that allow us to construct and split contracts. We use these maps to construct general maps:

6.10.1.1 Elementary maps.

Let $B_{\wedge}^{M}$ and $B_{\vee}^{M}$ be the monoids $B_{\wedge}^{M} = (B, \wedge, 1_B)$ and $B_{\vee}^{M} = (B, \vee, 0_B)$. We define the two monoid maps

$$\begin{aligned} \iota_a &\colon B_{\wedge}^{M} \to \mathcal{C}_{\|}^{M} \quad a \mapsto (a, 1_B) \\ \iota_g &\colon B_{\wedge}^{M} \to \mathcal{C}_{\|}^{M} \quad g \mapsto (1_B, g). \end{aligned}$$

These maps generate an epic monoid map $\pi \colon B_{\wedge}^{M} \times B_{\wedge}^{M} \to \mathcal{C}_{\|}^{M}(B)$ defined as

$$(a,g) \mapsto \iota_a(a) \parallel \iota_g(g) = (g \to a, g).$$

Similarly, we have monoid maps that allow us to split a contract:

$$\begin{aligned} \pi_g &\colon \mathcal{C}_{\|}^{M} \to B_{\wedge}^{M} \quad \pi_g(a,g) = g \\ \pi_a &\colon \mathcal{C}_{\wedge}^{M} \to B_{\vee}^{M} \quad \pi_a(a,g) = a. \end{aligned}$$

We use the monoid isomorphisms to obtain a map $\mathcal{C}_{\wedge}^{M} \to B_{\wedge}^{M}$ from the last morphism:

$$\begin{aligned} \neg \circ \pi_a \circ \theta_g &\colon \mathcal{C}_{\|}^{M} \to B_{\wedge}^{M} \\ (a,g) &\mapsto a \wedge g. \end{aligned}$$

The two maps $\mathcal{C}_{\|}^{M} \to B_{\wedge}^{M}$ yield the monic monoid map

$$\begin{aligned} \iota &\colon \mathcal{C}_{\|}^{M} \to B_{\wedge} \times B_{\wedge} \\ (a,g) &\mapsto (a \wedge g, g). \end{aligned}$$

This map is left-invertible:

$$\pi \circ \iota = \mathrm{id}.$$

6.10.1.2 General maps.

The elementary maps just described enable us to find the general structure between the monoid maps between the parallel monoids corresponding to two Boolean algebras. This is our main result in this section:

Theorem 6.10.3. *Let $f\colon \mathcal{C}_{\|}^{M}(B) \to \mathcal{C}_{\|}^{M}(B')$. There exists a unique $f^{\flat}\colon B_{\wedge}^{M} \times B_{\wedge}^{M} \to B_{\wedge}^{M'} \times B_{\wedge}^{M'}$ making the following diagram commute:*

$$\begin{array}{ccc} B_{\wedge}^{M} \times B_{\wedge}^{M} & \overset{f^{\flat}}{\dashrightarrow} & B_{\wedge}^{M'} \times B_{\wedge}^{M'} \\ {\scriptstyle\iota}\uparrow\downarrow{\scriptstyle\pi} & & {\scriptstyle\pi}\downarrow\uparrow{\scriptstyle\iota} \\ \mathcal{C}_{\|}^{M}(B) & \xrightarrow[f]{} & \mathcal{C}_{\|}^{M}(B') \end{array}$$

The structure of $f^{\flat}$ is given by

$$f^{\flat} = \left(l_a(ag)l_g(g)r_a(ag)r_g(g), r_a(ag)r_g(g)\right),$$

where $l_a, l_g, r_a, r_g\colon B_{\wedge}^{M} \to B_{\wedge}^{M'}$ are monoid morphisms.

Proof. Because ι is monic, f generates a unique monoid map $f^{\#}$:

$$\begin{array}{ccc} B_{\wedge}^{M} \times B_{\wedge}^{M} & & B_{\wedge}^{M'} \times B_{\wedge}^{M'} \\ \downarrow{\scriptstyle\pi} & \overset{f^{\#}}{\nearrow} & \uparrow{\scriptstyle\iota} \\ \mathcal{C}_{\|}^{M}(B) & \xrightarrow[f]{} & \mathcal{C}_{\|}^{M}(B') \end{array}$$

Because π is epic, $f^{\#}$ generates a unique monoid map $f^{\flat}$

$$\begin{array}{ccc} B_{\wedge}^{M} \times B_{\wedge}^{M} & \overset{f^{\flat}}{\dashrightarrow} & B_{\wedge}^{M'} \times B_{\wedge}^{M'} \\ \downarrow{\scriptstyle\pi} & \overset{f^{\#}}{\nearrow} & \uparrow{\scriptstyle\iota} \\ \mathcal{C}_{\|}^{M}(B) & \xrightarrow[f]{} & \mathcal{C}_{\|}^{M}(B') \end{array}$$

Thus, we have the diagram

$$\begin{array}{ccc} B_{\wedge}^{M} \times B_{\wedge}^{M} & \overset{f^{\flat}}{\dashrightarrow} & B_{\wedge}^{M'} \times B_{\wedge}^{M'} \\ {\scriptstyle\iota}\uparrow\downarrow{\scriptstyle\pi} & \overset{f^{\#}}{\nearrow} & {\scriptstyle\pi}\downarrow\uparrow{\scriptstyle\iota} \\ \mathcal{C}_{\|}^{M}(B) & \xrightarrow[f]{} & \mathcal{C}_{\|}^{M}(B') \end{array} \tag{6.4}$$

$f^{\flat}$ can be factored as the product of two maps

$$B_{\wedge}^{M} \times B_{\wedge}^{M} \to B_{\wedge}^{M'}.$$

We also observe that $f^\flat(a,g) = f^\flat((a,1)\wedge(1,g)) = f^\flat(a,1)\wedge f^\flat(1,g)$. This means there are monoid maps $l_a, l_g, r_a, r_g \colon B_\wedge^M \to B_\wedge^{M'}$ such that

$$f^\flat(a,g) = (l_a(a)l_g(g), r_a(a)r_g(g))\,.$$

To obtain further restrictions on these maps, we use (6.4):

$$\begin{aligned} f^\flat(a,g) &= \iota\circ f\circ\pi(a,g) = \iota\circ\pi\circ f^\flat\circ\iota\circ\pi(a,g)\\ &= (l_a(ag)l_g(g)r_a(ag)r_g(g), r_a(ag)r_g(g)) \end{aligned}$$

□

6.10.2 Semirings

Now that we have four monoids, we look for additional algebraic structure. First we study the distributivity of the binary operations.

	Conjunction	Disjunction	Composition	Merging
Conjunction	$\mathcal{C}\wedge(\mathcal{C}'\wedge\mathcal{C}'') =$ $(\mathcal{C}\wedge\mathcal{C}')\wedge(\mathcal{C}\wedge\mathcal{C}'')$	$\mathcal{C}\wedge(\mathcal{C}'\vee\mathcal{C}'') =$ $(\mathcal{C}\wedge\mathcal{C}')\vee(\mathcal{C}\wedge\mathcal{C}'')$	$\mathcal{C}\wedge(\mathcal{C}'\parallel\mathcal{C}'') =$ $(\mathcal{C}\wedge\mathcal{C}')\parallel(\mathcal{C}\wedge\mathcal{C}'')$	$\mathrm{id}\wedge(1\bullet 0)\neq$ $(\mathrm{id}\wedge 1)\bullet(\mathrm{id}\wedge 0)$
Disjunction	$\mathcal{C}\vee(\mathcal{C}'\wedge\mathcal{C}'') =$ $(\mathcal{C}\vee\mathcal{C}')\wedge(\mathcal{C}\vee\mathcal{C}'')$	$\mathcal{C}\vee(\mathcal{C}'\vee\mathcal{C}'') =$ $(\mathcal{C}\vee\mathcal{C}')\vee(\mathcal{C}\vee\mathcal{C}'')$	$\mathrm{id}\vee(1\parallel 0)\neq$ $(\mathrm{id}\vee 1)\parallel(\mathrm{id}\vee 0)$	$\mathcal{C}\vee(\mathcal{C}'\bullet\mathcal{C}'') =$ $(\mathcal{C}\vee\mathcal{C}')\bullet(\mathcal{C}\vee\mathcal{C}'')$
Compostion	$\mathcal{C}\parallel(\mathcal{C}'\wedge\mathcal{C}'') =$ $(\mathcal{C}\parallel\mathcal{C}')\wedge(\mathcal{C}\parallel\mathcal{C}'')$	$\mathcal{C}\parallel(\mathcal{C}'\vee\mathcal{C}'') =$ $(\mathcal{C}\parallel\mathcal{C}')\vee(\mathcal{C}\parallel\mathcal{C}'')$	$\mathcal{C}\parallel(\mathcal{C}'\parallel\mathcal{C}'') =$ $(\mathcal{C}\parallel\mathcal{C}')\parallel(\mathcal{C}\parallel\mathcal{C}'')$	$1\parallel(0\bullet\mathrm{id})\neq$ $(1\parallel 0)\bullet(1\parallel\mathrm{id})$
Merging	$\mathcal{C}\bullet(\mathcal{C}'\wedge\mathcal{C}'') =$ $(\mathcal{C}\bullet\mathcal{C}')\wedge(\mathcal{C}\bullet\mathcal{C}'')$	$\mathcal{C}\bullet(\mathcal{C}'\vee\mathcal{C}'') =$ $(\mathcal{C}\bullet\mathcal{C}')\vee(\mathcal{C}\bullet\mathcal{C}'')$	$0\bullet(1\parallel\mathrm{id})\neq$ $(0\bullet 1)\parallel(0\bullet\mathrm{id})$	$\mathcal{C}\bullet(\mathcal{C}'\bullet\mathcal{C}'') =$ $(\mathcal{C}\bullet\mathcal{C}')\bullet(\mathcal{C}\bullet\mathcal{C}'')$

Table 6.5: Distributivity of contract operations

Proposition 6.10.4. *Table 6.5 shows whether the binary operations displayed in the rows distribute over the binary operations in the columns.*

Proof. Let $\mathcal{C} = (a,g)$, $\mathcal{C}' = (a',g')$, and $\mathcal{C}'' = (a'',g'')$ be contracts.

- Conjunction.

$$\begin{aligned}
\mathcal{C} \wedge (\mathcal{C}' \vee \mathcal{C}'') &= (a, g) \wedge (a' \wedge a'', g' \vee g'') = ((a \vee a') \wedge (a \vee a''), (g \wedge g') \vee (g \wedge g'')) \\
&= (\mathcal{C} \wedge \mathcal{C}') \vee (\mathcal{C} \wedge \mathcal{C}'') \\
\mathcal{C} \wedge (\mathcal{C}' \parallel \mathcal{C}'') &= (a, g) \wedge ((g' \wedge g'') \rightarrow (a' \wedge a''), g' \wedge g'') \\
&= (a \vee (a' \wedge a'') \vee \neg g' \vee \neg g'', g \wedge g' \wedge g'') \\
&= (a \vee \neg g \vee (a' \wedge a'') \vee \neg g' \vee \neg g'', g \wedge g' \wedge g'') \\
&= ((g \wedge g' \wedge g'') \rightarrow ((a \vee a') \wedge (a \vee a'')), g \wedge g' \wedge g'') \\
&= (a \vee a', g \wedge g') \parallel (a \vee a'', g \wedge g'') = (\mathcal{C} \wedge \mathcal{C}') \parallel (\mathcal{C} \wedge \mathcal{C}'') \\
\mathrm{id} \wedge (1 \bullet 0) &= \mathrm{id} \wedge 1 = \mathrm{id} \neq 0 = \mathrm{id} \bullet 0 = (\mathrm{id} \wedge 1) \bullet (\mathrm{id} \wedge 0)
\end{aligned}$$

- Disjunction.

$$\begin{aligned}
\mathcal{C} \vee (\mathcal{C}' \wedge \mathcal{C}'') &= \left(\mathcal{C}^{-1} \wedge ((\mathcal{C}')^{-1} \vee (\mathcal{C}'')^{-1})\right)^{-1} \\
&= \left((\mathcal{C}^{-1} \wedge (\mathcal{C}')^{-1}) \vee (\mathcal{C}^{-1} \wedge (\mathcal{C}'')^{-1})\right)^{-1} \\
&= (\mathcal{C} \vee \mathcal{C}') \wedge (\mathcal{C} \vee \mathcal{C}'') \\
\mathcal{C} \vee (\mathcal{C}' \bullet \mathcal{C}'') &= \left(\mathcal{C}^{-1} \wedge ((\mathcal{C}')^{-1} \parallel (\mathcal{C}'')^{-1})\right)^{-1} \\
&= \left((\mathcal{C}^{-1} \wedge (\mathcal{C}')^{-1}) \parallel (\mathcal{C}^{-1} \wedge (\mathcal{C}'')^{-1})\right)^{-1} \\
&= (\mathcal{C} \vee \mathcal{C}') \bullet (\mathcal{C} \vee \mathcal{C}'') \\
\mathrm{id} \vee (1 \parallel 0) &= \mathrm{id} \vee 0 = \mathrm{id} \neq 1 = 1 \parallel \mathrm{id} = (\mathrm{id} \vee 1) \parallel (\mathrm{id} \vee 0)
\end{aligned}$$

- Composition.

$$\begin{aligned}
\mathcal{C} \parallel (\mathcal{C}' \wedge \mathcal{C}'') &= (a, g) \parallel (a' \vee a'', g' \wedge g'') \\
&= (\neg(g \wedge g') \vee \neg(g \wedge g'') \vee (a \wedge a') \vee (a \wedge a''), (g \wedge g') \wedge (g \wedge g'')) \\
&= ((g \wedge g') \rightarrow (a \wedge a'), (g \wedge g')) \wedge ((g \wedge g'') \rightarrow (a \wedge a''), (g \wedge g'')) \\
&= (\mathcal{C} \parallel \mathcal{C}') \wedge (\mathcal{C} \parallel \mathcal{C}'') \\
\mathcal{C} \parallel (\mathcal{C}' \vee \mathcal{C}'') &= (a, g) \parallel (a' \wedge a'', g' \vee g'') \\
&= (\neg(g \wedge g') \wedge \neg(g \wedge g'') \vee ((a \wedge a') \wedge (a \wedge a'')), (g \wedge g') \vee (g \wedge g'')) \\
&= ((\neg(g \wedge g') \vee (a \wedge a')) \wedge (\neg(g \wedge g'') \vee (a \wedge a'')), (g \wedge g') \vee (g \wedge g'')) \\
&= ((g \wedge g') \rightarrow (a \wedge a'), (g \wedge g')) \vee ((g \wedge g'') \rightarrow (a \wedge a''), (g \wedge g'')) \\
&= (\mathcal{C} \parallel \mathcal{C}') \vee (\mathcal{C} \parallel \mathcal{C}'') \\
1 \parallel (0 \bullet \mathrm{id}) &= 1 \parallel 0 = 0 \neq 1 = 0 \bullet 1 = (1 \parallel 0) \bullet (1 \parallel \mathrm{id})
\end{aligned}$$

- Merging.

$$\begin{aligned}
\mathcal{C} \bullet (\mathcal{C}' \wedge \mathcal{C}'') &= \left(\mathcal{C}^{-1} \parallel ((\mathcal{C}')^{-1} \vee (\mathcal{C}'')^{-1})\right)^{-1} \\
&= \left((\mathcal{C}^{-1} \parallel (\mathcal{C}')^{-1}) \vee (\mathcal{C}^{-1} \parallel (\mathcal{C}'')^{-1})\right)^{-1} \\
&= (\mathcal{C} \bullet \mathcal{C}') \wedge (\mathcal{C} \bullet \mathcal{C}'') \\
\mathcal{C} \bullet (\mathcal{C}' \vee \mathcal{C}'') &= \left(\mathcal{C}^{-1} \parallel ((\mathcal{C}')^{-1} \wedge (\mathcal{C}'')^{-1})\right)^{-1} \\
&= \left((\mathcal{C}^{-1} \parallel (\mathcal{C}')^{-1}) \wedge (\mathcal{C}^{-1} \parallel (\mathcal{C}'')^{-1})\right)^{-1} \\
&= (\mathcal{C} \bullet \mathcal{C}') \vee (\mathcal{C} \bullet \mathcal{C}'') \\
0 \bullet (1 \parallel \mathrm{id}) &= 0 \bullet 1 = 1 \neq 0 = 1 \parallel 0 = (0 \bullet 1) \parallel (0 \bullet \mathrm{id})
\end{aligned}$$

□

One can find a notion of sub-distributivity for contracts in [19]. The distributivity of composition over conjunction can be found in [144].

We will use the distributivity results to look for semiring structure within the algebra of contracts. We recall the definition of a semiring (see, e.g., [73]):

Definition 6.10.5. *A semiring $(R, \cdot, +, 1_R, 0_R)$ is a nonempty set R with operations of multiplication and addition satisfying*

a. $(R, +, 0_R)$ *is a commutative monoid*

b. $(R, \cdot, 1_R)$ *is a monoid*

c. $r(s+t) = rs + rt$ *and* $(s+t)r = sr + tr$ *for all* $r, s, t \in R$

d. $r \cdot 0_R = 0_R \cdot r = 0_R$ *for all* $r \in R$

e. $0_R \neq 1_R$.

A map of semirings $f : (R, \cdot, +, 1_R, 0_R) \to (R', \cdot, +, 1_{R'}, 0_{R'})$ *satisfies*

- $f(0_R) = f(0_{R'})$
- $f(1_R) = f(1_{R'})$
- $f(r+s) = f(r) + f(s)$
- $f(r \cdot s) = f(r) \cdot f(s)$.

The following result provides the semiring structures available in the contract algebra.

Proposition 6.10.6. *Using the operations of conjunction, disjunction, composition, and merging, we have exactly four semirings:*

- *The conjunction semiring.* $\mathcal{C}_\wedge(B) = (\mathcal{C}(B), \wedge, \vee, 1, 0)$
- *The disjunction semiring.* $\mathcal{C}_\vee(B) = (\mathcal{C}(B), \vee, \wedge, 0, 1)$
- *The composition semiring.* $\mathcal{C}_\parallel(B) = (\mathcal{C}(B), \parallel, \vee, id, 0)$
- *The merging semiring.* $\mathcal{C}_\bullet(B) = (\mathcal{C}(B), \bullet, \wedge, id, 1)$

Proof. Tables 6.4 and 6.5 tell, respectively, how operations behave with respect to the distinguished elements and how operations distribute.

Suppose conjunction is the multiplication operation. Since $\mathcal{C} \wedge \mathrm{id} \neq \mathrm{id}$, neither merging nor composition can be the addition operations. On the other hand, $\mathcal{C} \wedge 0 = 0$, and conjunction distributes over disjunction. Thus, $(\mathcal{C}(B), \wedge, \vee, 1, 0)$ is a semiring.

Now we assume disjunction is the multiplication operation. Since $\mathcal{C} \vee \mathrm{id} \neq \mathrm{id}$, neither merging nor composition can be the addition operations. However, $\mathcal{C} \vee 1 = 1$, and disjunction distributes over conjunction. Thus, $(\mathcal{C}(B), \vee, \wedge, 0, 1)$ is a semiring.

Suppose composition is the multiplication operation. Since composition does not distribute over merging, merging cannot be addition. Since $\mathcal{C} \parallel 1 \neq 1$, conjunction cannot be addition. However, $\mathcal{C} \parallel 0 = 0$ and composition distributes over disjunction. Thus, $(\mathcal{C}(B), \parallel, \vee, \mathrm{id}, 0)$ is a semiring.

Now suppose that merging is the multiplication. Since merging does not distribute over composition, composition cannot be addition. Also, since $\mathcal{C} \bullet 0 \neq 0$, conjunction cannot be addition. However, $\mathcal{C} \bullet 1 = 1$ and merging distributes over conjunction. Thus, $(\mathcal{C}(B), \bullet, \wedge, \mathrm{id}, 1)$ is a semiring. □

The reciprocal map of contracts generates the following isomorphisms:

- The conjunction and disjunction semirings are isomorphic.
- The composition and merging semirings are isomorphic.

Let $B_\wedge$ and $B_\vee$ be, respectively, the semirings $(B, \wedge, \vee, 1, 0)$ and $(B, \vee, \wedge, 0, 1)$. We first observe that complementation is a semiring isomorphism for $B_\wedge$ and $B_\vee$. We define maps $\Delta_g : B_\wedge \to \mathcal{C}_\wedge$ and $\iota_g : B_\wedge \to \mathcal{C}_\parallel$ as follows:

$$\Delta_g(b) = (\neg b, b), \text{ and}$$
$$\iota_g(b) = (1_B, b).$$

Proposition 6.10.7. *Δ_g and ι_g are semiring homomorphisms.*

Proof. Let $b, b' \in B$.

- $\Delta_g(0_B) = (1_B, 0_B) = 0$ and $\Delta_g(1_B) = (0_B, 1_B) = 1$
- $\Delta_g(b \wedge b') = (\neg(b \wedge b'), b \wedge b') = (\neg b \vee \neg b', b \wedge b') = \Delta_g(b) \wedge \Delta_g(b')$

- $\Delta_g(b \vee b') = (\neg(b \vee b'), b \vee b') = (\neg b \wedge \neg b', b \vee b') = \Delta_g(b) \vee \Delta_g(b')$.

This shows that Δ_g is a semiring homomorphism. Now we study ι_g:

- $\iota_g(0_B) = (1_B, 0_B) = 0$ and $\iota_g(1_B) = (1_B, 1_B) = \mathrm{id}$
- $\iota_g(b \wedge b') = (1_B, b \wedge b') = (1_B, b) \parallel (1_B, b') = \iota_g(b) \parallel \iota_g(b')$
- $\iota_g(b \vee b') = (1, b \vee b') = (1_B, b) \vee (1_B, b') = \iota_g(b) \vee \iota_g(b')$.

We conclude that ι_g is a semiring homomorphism as well. □

Observe that Δ_g can be used to obtain a semiring map from $B_\vee$ to $\mathcal{C}_\vee$ using the semiring isomorphisms $\neg : B_\wedge \xrightarrow{\sim} B_\vee$ and $(\cdot)^{-1} : \mathcal{C}_\wedge(B) \xrightarrow{\sim} \mathcal{C}_\vee(B)$ as follows:

$$B_\vee \xrightarrow{\neg} B_\wedge \xrightarrow{\Delta_g} \mathcal{C}_\wedge(B) \xrightarrow{(\cdot)^{-1}} \mathcal{C}_\vee(B)$$

$$b \longmapsto \neg b \longmapsto (b, \neg b) \longmapsto (\neg b, b)$$

This means that Δ_g is also a semiring homomorphism $B_\vee \xrightarrow{\Delta_g} \mathcal{C}_\vee(B)$. The following diagram commutes in the category of semirings:

$$\begin{array}{ccc} B_\wedge & \underset{\simeq}{\overset{\neg}{\longleftrightarrow}} & B_\vee \\ {\scriptstyle \Delta_g}\downarrow & \swarrow\ {\scriptstyle \Delta_a}\ \searrow & \downarrow{\scriptstyle \Delta_g} \\ \mathcal{C}_\wedge(B) & \underset{(\cdot)^{-1}}{\overset{\simeq}{\longleftrightarrow}} & \mathcal{C}_\vee(B) \end{array}$$

The commutativity of the diagram gives rise to the diagonal arrow $\Delta_a = (\cdot)^{-1} \circ \Delta_g = \Delta_g \circ \neg$. This map is a semiring homomorphism from $B_\wedge$ to $\mathcal{C}_\vee(B)$ and from $B_\vee$ to $\mathcal{C}_\wedge(B)$. Explicitly, this map is

$$\Delta_a b = (\Delta_g(b))^{-1} = (\neg b, b)^{-1} = (b, \neg b) \quad (b \in B).$$

Now, if we use the map ι_g, we can obtain a map $\iota'_a : B_\vee \to \mathcal{C}_\bullet(B)$ as follows:

$$B_\vee \xrightarrow{\neg} B_\wedge \xrightarrow{\iota_g} \mathcal{C}_\parallel(B) \xrightarrow{(\cdot)^{-1}} \mathcal{C}_\vee(B)$$

$$b \longmapsto \neg b \longmapsto (1_B, \neg b) \longmapsto (\neg b, 1_B)$$

We obtain the diagram below.

$$\begin{array}{ccc} B_\wedge & \underset{\simeq}{\overset{\neg}{\longleftrightarrow}} & B_\vee \\ {\scriptstyle \iota_g}\downarrow & \swarrow\ {\scriptstyle \iota_a} \quad {\scriptstyle \iota'_g}\ \searrow & \downarrow{\scriptstyle \iota'_a} \\ \mathcal{C}_\parallel(B) & \underset{(\cdot)^{-1}}{\overset{\simeq}{\longleftrightarrow}} & \mathcal{C}_\bullet(B) \end{array}$$

The commutativity of the diagram provides the semiring maps $B_\wedge \xrightarrow{\iota_a} \mathcal{C}_\bullet(B)$ and $B_\vee \xrightarrow{\iota'_g} \mathcal{C}_\parallel(B)$ given by $\iota_a = (\cdot)^{-1} \circ \iota_g$ and $\iota'_g = \iota_g \circ \neg$.

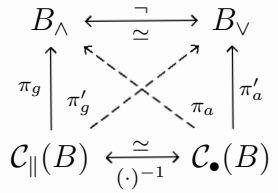
Now we consider maps to $B_\wedge$. The map

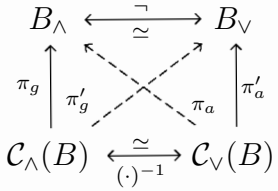
$$\pi_g : (a, g) \mapsto g$$

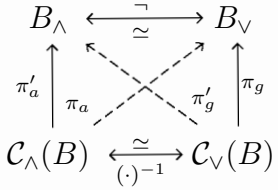
is a semiring homomorphism from $\mathcal{C}_\wedge(B)$ to $B_\wedge$ and from $\mathcal{C}_\parallel(B)$ to $B_\wedge$. Similarly, the map

$$\pi'_a : (a, g) \mapsto \neg a$$

is a semiring homomorphism from $\mathcal{C}_\wedge(B)$ to $B_\wedge$. The following diagrams commute and define the maps not specified before.







6.11 Actions

The semiring maps just described can be used to generate actions of the semirings $B_\wedge$ and $B_\vee$ over the contract semirings. Consider, for example the map $\Delta_g : B_\wedge \to \mathcal{C}_\wedge(B)$. For a contract $\mathcal{C} = (a, g)$, we have

$$\Delta_g(b) \wedge \mathcal{C} = (\neg b, b) \wedge (a, g) = (b \to a, b \wedge g).$$

Now consider the map $\iota_g : B_\wedge \to \mathcal{C}_\parallel(B)$:

$$\iota_g(b) \parallel \mathcal{C} = (1_B, b) \parallel (a, g) = ((b \wedge g) \to a, b \wedge g) = (b \to a, b \wedge g).$$

Order $b \cdot \mathcal{C} \geq \mathcal{C}$	$\mathcal{C} \cdot b \leq \mathcal{C}$
Reciprocal $(b \cdot \mathcal{C})^{-1} = \mathcal{C}^{-1} \cdot b$	
Associativity $(b \wedge b') \cdot \mathcal{C} = b \cdot (b' \cdot \mathcal{C})$	$\mathcal{C} \cdot (b \wedge b') = (\mathcal{C} \cdot b) \cdot b'$
Distributivity over the Boolean algebra $(b \vee b') \cdot \mathcal{C} = (b \cdot \mathcal{C}) \wedge (b' \cdot \mathcal{C})$	$\mathcal{C} \cdot (b \vee b') = (\mathcal{C} \cdot b) \vee (\mathcal{C} \cdot b')$
Actions and the contract operations $b \cdot (\mathcal{C} \wedge \mathcal{C}') = b \cdot \mathcal{C} \wedge b \cdot \mathcal{C}'$ $b \cdot (\mathcal{C} \vee \mathcal{C}') = b \cdot \mathcal{C} \vee \mathcal{C}'$ $b \cdot (\mathcal{C} \parallel \mathcal{C}') = b \cdot \mathcal{C} \parallel b \cdot \mathcal{C}'$ $b \cdot (\mathcal{C} \bullet \mathcal{C}') = b \cdot \mathcal{C} \bullet \mathcal{C}'$	 $(\mathcal{C} \wedge \mathcal{C}') \cdot b = \mathcal{C} \cdot b \wedge \mathcal{C}'$ $(\mathcal{C} \vee \mathcal{C}') \cdot b = \mathcal{C} \cdot b \vee \mathcal{C}' \cdot b$ $(\mathcal{C} \parallel \mathcal{C}') \cdot b = \mathcal{C} \cdot b \parallel \mathcal{C}'$ $(\mathcal{C} \bullet \mathcal{C}') \cdot b = \mathcal{C} \cdot b \bullet \mathcal{C}' \cdot b$
Actions and the adjoint operations $b \cdot (\mathcal{C}/\mathcal{C}') = \mathcal{C}/(\mathcal{C}' \cdot b) = (b \cdot \mathcal{C})/\mathcal{C}'$ $b \cdot (\mathcal{C} \div \mathcal{C}') = (b \cdot \mathcal{C}) \div (\mathcal{C}' \cdot b)$ $b \cdot (\mathcal{C}' \to \mathcal{C}) = \mathcal{C}' \to b \cdot \mathcal{C} = \mathcal{C}' \cdot b \to \mathcal{C}$ $b \cdot (\mathcal{C}' \nrightarrow \mathcal{C}) = \mathcal{C}' \cdot b \nrightarrow b \cdot \mathcal{C}$	 $(\mathcal{C}/\mathcal{C}') \cdot b = (\mathcal{C} \cdot b)/(b \cdot \mathcal{C}')$ $(\mathcal{C} \div \mathcal{C}') \cdot b = (\mathcal{C} \cdot b) \div \mathcal{C}' = \mathcal{C} \div (b \cdot \mathcal{C}')$ $(\mathcal{C}' \to \mathcal{C}) \cdot b = b \cdot \mathcal{C}' \to \mathcal{C} \cdot b$ $(\mathcal{C}' \nrightarrow \mathcal{C}) \cdot b = \mathcal{C}' \nrightarrow \mathcal{C} \cdot b = b \cdot \mathcal{C}' \nrightarrow \mathcal{C}$

Table 6.6: Identities for the left and right actions of a Boolean algebra B over its contract algebra ($b, b' \in B$ and $\mathcal{C}, \mathcal{C}' \in \mathcal{C}(B)$)

We observe that elements of B act in the same way on the semirings $\mathcal{C}_\wedge(B)$ and $\mathcal{C}_\parallel(B)$. We define the right action of B on $\mathcal{C}(B)$ as

$$(a, g) \cdot b = (b \to a, b \wedge g). \tag{6.5}$$

Similarly, the semiring map $\Delta_a : B_\wedge \to \mathcal{C}_\vee(B)$ yields

$$\Delta_a(b) \vee \mathcal{C} = (b, \neg b) \vee (a, g) = (b \wedge a, b \to g),$$

and the semiring map $\iota_a : B_\wedge \to \mathcal{C}_\bullet(B)$ results in

$$\iota_a(b) \bullet \mathcal{C} = (b, 1_B) \bullet (a, g) = (b \wedge a, b \to g).$$

We thus define the left action of B on $\mathcal{C}(B)$ as

$$b \cdot (a, g) = (b \wedge a, b \to g). \tag{6.6}$$

The following proposition shows several properties of these actions.

Proposition 6.11.1. *The identities for the left and right actions shown in Table 6.6 hold.*

Proof. Let $b, b' \in B$, $\mathcal{C} = (a, g)$, and $\mathcal{C}' = (a', g')$. We have the following properties:

- Order.

$$b \cdot \mathcal{C} = b \cdot (a, g) = (b \wedge a, b \to g) \geq (a, g) = \mathcal{C}$$
$$\mathcal{C} \cdot b = (a, g) \cdot b = (b \to a, b \wedge g) \leq (a, g) = \mathcal{C}$$

- Reciprocal.

$$(b \cdot \mathcal{C})^{-1} = (b \wedge a, b \to g)^{-1} = (b \to g, b \wedge a) = (g, a) \cdot b = \mathcal{C}^{-1} \cdot b$$

- Associativity.

$$\begin{aligned}(b \wedge b') \cdot \mathcal{C} &= ((b \wedge b') \wedge a, (b \wedge b') \to g) = (b \wedge (b' \wedge a), b \to (b' \to g)) = b \cdot (b' \cdot (a, g)) \\ &= b \cdot (b' \cdot \mathcal{C}) \\ \mathcal{C} \cdot (b \wedge b') &= \left((b \wedge b') \cdot \mathcal{C}^{-1}\right)^{-1} = \left((b' \wedge b) \cdot \mathcal{C}^{-1}\right)^{-1} = \left(b' \cdot \left(b \cdot \mathcal{C}^{-1}\right)\right)^{-1} \\ &= \left(b \cdot \mathcal{C}^{-1}\right)^{-1} \cdot b' = (\mathcal{C} \cdot b) \cdot b'\end{aligned}$$

- Distributivity over the Boolean algebra B.

$$\begin{aligned}(b \vee b') \cdot \mathcal{C} &= ((b \vee b') \wedge a, (b \vee b') \to g) = ((b \wedge a) \vee (b' \wedge a), (b \to g) \wedge (b' \to g)) \\ &= (b \cdot \mathcal{C}) \wedge (b' \cdot \mathcal{C}) \\ \mathcal{C} \cdot (b \vee b') &= \left((b \vee b') \cdot \mathcal{C}^{-1}\right)^{-1} = \left(b \cdot \mathcal{C}^{-1} \wedge b' \cdot \mathcal{C}^{-1}\right)^{-1} = \mathcal{C} \cdot b \vee \mathcal{C} \cdot b'\end{aligned}$$

- Distributivity over the contract operations.
  - Conjunction.

$$\begin{aligned}b \cdot (\mathcal{C} \wedge \mathcal{C}') &= (b \wedge (a \vee a'), b \to (g \wedge g')) = ((b \wedge a) \vee (b \wedge a'), (b \to g) \wedge (b \to g')) \\ &= b \cdot \mathcal{C} \wedge b \cdot \mathcal{C}' \\ (\mathcal{C} \wedge \mathcal{C}') \cdot b &= (b \to (a \vee a'), b \wedge (g \wedge g')) = ((b \to a) \vee a', (b \wedge g) \wedge g') \\ &= (\mathcal{C} \cdot b) \wedge \mathcal{C}'\end{aligned}$$

  - Disjunction.

$$b \cdot (\mathcal{C} \vee \mathcal{C}') = \left(\left(\mathcal{C}^{-1} \wedge (\mathcal{C}')^{-1}\right) \cdot b\right)^{-1} = \left(\mathcal{C}^{-1} \cdot b \wedge (\mathcal{C}')^{-1}\right)^{-1} = b \cdot \mathcal{C} \vee \mathcal{C}'$$
$$(\mathcal{C} \vee \mathcal{C}') \cdot b = \left(b \cdot \left(\mathcal{C}^{-1} \wedge (\mathcal{C}')^{-1}\right)\right)^{-1} = \left(b \cdot \mathcal{C}^{-1} \wedge b \cdot (\mathcal{C}')^{-1}\right)^{-1} = \mathcal{C} \cdot b \vee \mathcal{C}' \cdot b$$

– Composition.

$$\begin{aligned}
b \cdot (\mathcal{C} \parallel \mathcal{C}') &= (b \wedge ((g \wedge g') \rightarrow (a \wedge a')), b \rightarrow (g \wedge g')) \\
&= ((b \rightarrow (g \wedge g')) \rightarrow (b \wedge a \wedge a'), b \rightarrow (g \wedge g')) \\
&= (((b \rightarrow g) \wedge (b \rightarrow g')) \rightarrow ((b \wedge a) \wedge (b \wedge a')), (b \rightarrow g) \wedge (b \rightarrow g')) \\
&= b \cdot \mathcal{C} \parallel b \cdot \mathcal{C}' \\
(\mathcal{C} \parallel \mathcal{C}') \cdot b &= (b \rightarrow ((g \wedge g') \rightarrow (a \wedge a')), b \wedge g \wedge g') \\
&= ((b \wedge g \wedge g') \rightarrow (a \wedge a'), b \wedge g \wedge g') \\
&= ((b \wedge g \wedge g') \rightarrow ((b \rightarrow a) \wedge a'), b \wedge g \wedge g') \\
&= (\mathcal{C} \cdot b) \parallel \mathcal{C}'
\end{aligned}$$

– Merging.

$$\begin{aligned}
b \cdot (\mathcal{C} \bullet \mathcal{C}') &= \left((\mathcal{C}^{-1} \parallel (\mathcal{C}')^{-1}) \cdot b\right)^{-1} = \left(\mathcal{C}^{-1} \cdot b \parallel (\mathcal{C}')^{-1}\right)^{-1} = b \cdot \mathcal{C} \bullet \mathcal{C}' \\
(\mathcal{C} \bullet \mathcal{C}') \cdot b &= \left(b \cdot (\mathcal{C}^{-1} \parallel (\mathcal{C}')^{-1})\right)^{-1} = \left(b \cdot \mathcal{C}^{-1} \parallel b \cdot (\mathcal{C}')^{-1}\right)^{-1} = \mathcal{C} \cdot b \bullet \mathcal{C}' \cdot b
\end{aligned}$$

- Distributivity over the adjoint operations.

– Quotient.

$$\begin{aligned}
b \cdot (\mathcal{C}/\mathcal{C}') &= b \cdot (\mathcal{C} \bullet (\mathcal{C}')^{-1}) = b \cdot \mathcal{C} \bullet (\mathcal{C}')^{-1} \\
&= (b \cdot \mathcal{C})/(\mathcal{C}') \\
b \cdot (\mathcal{C}/\mathcal{C}') &= b \cdot (\mathcal{C} \bullet (\mathcal{C}')^{-1}) = \mathcal{C} \bullet b \cdot (\mathcal{C}')^{-1} = \mathcal{C} \bullet (\mathcal{C}' \cdot b)^{-1} \\
&= \mathcal{C}/(\mathcal{C}' \cdot b) \\
(\mathcal{C}/\mathcal{C}') \cdot b &= (\mathcal{C} \bullet (\mathcal{C}')^{-1}) \cdot b = \mathcal{C} \cdot b \bullet (\mathcal{C}')^{-1} \cdot b = \mathcal{C} \cdot b \bullet (b \cdot \mathcal{C}')^{-1} \\
&= (\mathcal{C} \cdot b)/(b \cdot \mathcal{C}')
\end{aligned}$$

– Separation.

$$\begin{aligned}
b \cdot (\mathcal{C} \div \mathcal{C}') &= b \cdot (\mathcal{C} \parallel (\mathcal{C}')^{-1}) = b \cdot \mathcal{C} \parallel b \cdot (\mathcal{C}')^{-1} = b \cdot \mathcal{C} \parallel (\mathcal{C}' \cdot b)^{-1} \\
&= (b \cdot \mathcal{C}) \div (\mathcal{C}' \cdot b) \\
(\mathcal{C} \div \mathcal{C}') \cdot b &= (\mathcal{C} \parallel (\mathcal{C}')^{-1}) \cdot b = \mathcal{C} \cdot b \parallel (\mathcal{C}')^{-1} = (\mathcal{C} \cdot b) \div \mathcal{C}' \\
(\mathcal{C} \div \mathcal{C}') \cdot b &= (\mathcal{C} \parallel (\mathcal{C}')^{-1}) \cdot b = \mathcal{C} \parallel (\mathcal{C}')^{-1} \cdot b = \mathcal{C} \div (b \cdot \mathcal{C}')
\end{aligned}$$

– Implication.

$$\begin{aligned}
b \cdot (\mathcal{C}' \rightarrow \mathcal{C}) &= b \cdot ((a \wedge \neg a') \vee (g' \wedge \neg g), g \vee \neg g') \\
&= (b \wedge (a \wedge \neg a') \vee (b \wedge g' \wedge \neg g), (b \rightarrow g) \vee \neg g') \\
&= (((b \wedge a) \wedge \neg a') \vee (g' \wedge \neg (b \rightarrow g)), (b \rightarrow g) \vee \neg g')
\end{aligned}$$

$$
\begin{aligned}
&= \mathcal{C}' \to b \cdot \mathcal{C} \\
b \cdot (\mathcal{C}' \to \mathcal{C}) &= b \cdot ((a \wedge \neg a') \vee (g' \wedge \neg g), g \vee \neg g') \\
&= ((a \wedge \neg(b \to a')) \vee (b \wedge g' \wedge \neg g), (b \wedge g') \to g) \\
&= \mathcal{C}' \cdot b \to \mathcal{C} \\
(\mathcal{C}' \to \mathcal{C}) \cdot b &= ((a \wedge \neg a') \vee (g' \wedge \neg g), g \vee \neg g') \cdot b \\
&= ((b \to (a \wedge \neg a')) \vee (b \to (g' \wedge \neg g)), (b \wedge g) \vee \neg(b \to g')) \\
&= ((b \to a) \wedge \neg(b \wedge a') \vee (b \to g') \wedge \neg(b \wedge g), (b \wedge g) \vee \neg(b \to g')) \\
&= b \cdot \mathcal{C}' \to \mathcal{C} \cdot b
\end{aligned}
$$

– Coimplication.

$$
\begin{aligned}
b \cdot (\mathcal{C}' \nrightarrow \mathcal{C}) &= \left(((\mathcal{C}')^{-1} \to \mathcal{C}^{-1}) \cdot b\right)^{-1} = \left(b \cdot (\mathcal{C}')^{-1} \to \mathcal{C}^{-1} \cdot b\right)^{-1} \\
&= \mathcal{C}' \cdot b \nrightarrow b \cdot \mathcal{C} \\
(\mathcal{C}' \nrightarrow \mathcal{C}) \cdot b &= \left(b \cdot ((\mathcal{C}')^{-1} \to \mathcal{C}^{-1})\right)^{-1} \\
&= \left((\mathcal{C}')^{-1} \to b \cdot \mathcal{C}^{-1}\right)^{-1} = \mathcal{C}' \nrightarrow \mathcal{C} \cdot b \\
&= \left((\mathcal{C}')^{-1} \cdot b \to \mathcal{C}^{-1}\right)^{-1} = b \cdot \mathcal{C}' \nrightarrow \mathcal{C}
\end{aligned}
$$

□

6.12 Contract abstractions

It is often useful to vary the level of detail used to represent objects. More detail may be needed to carry out some analysis tasks, but too much detail may hinder computation. We will explore abstract interpretation for contracts.

First, suppose we have two Boolean algebras B and B' and a Boolean algebra map $f : B \to B'$. f induces a map $f^* : \mathcal{C}(B) \to \mathcal{C}(B')$ between their contract algebras given by

$$f^*(a, g) = (f(a), f(g)).$$

Observe that $f(a) \vee f(g) = f(a \vee g) = f(1_B) = 1_{B'}$. As f commutes with the Boolean algebra operations, f^* commutes with the contract operations. Thus f^* is well-defined.

Suppose we have abstract and concrete domains given by Boolean algebras B_a and B_c, respectively. Moreover, suppose there are Boolean algebra maps $\gamma : B_a \to B_c$ and $\alpha : B_c \to B_a$ which we will call concretization and abstraction, respectively. These maps generate contract maps

$$\mathcal{C}(B_a) \xrightarrow{\gamma^*} \mathcal{C}(B_c) \quad \text{and} \quad \mathcal{C}(B_c) \xrightarrow{\alpha^*} \mathcal{C}(B_a),$$

as described before.

We are interested in exploring the conditions needed for these maps to form a Galois connection. Specifically, for $\mathcal{C}_a = (a_a, g_a) \in \mathcal{C}(B_a)$ and $\mathcal{C}_c = (a_c, g_c) \in \mathcal{C}(B_c)$, we want

$$\alpha^*(\mathcal{C}_c) \leq \mathcal{C}_a \text{ if and only if } \mathcal{C}_c \leq \gamma^*(\mathcal{C}_a).$$

This means that

$$(\alpha a_c, \alpha g_c) \leq (a_a, g_a) \text{ if and only if } (a_c, g_c) \leq (\gamma a_a, \gamma g_a).$$

If we set $a_c = 1_{B_c}$ and $a_a = 1_{B_a}$, we get

$$\alpha g_c \leq g_a \text{ if and only if } g_c \leq \gamma g_a,$$

and if we set $g_c = 1_{B_c}$ and $g_a = 1_{B_a}$, we obtain

$$a_a \leq \alpha a_c \text{ if and only if } \gamma a_a \leq a_c.$$

This means that α and γ must be simultaneously the left and right adjoints of each other. By setting $g_c = \gamma g_a$ and $a_c = \gamma a_a$ in the equations above, we obtain that $\alpha \circ \gamma$ is the identity map. Similarly, by setting $g_a = \alpha g_c$ and $a_a = \alpha a_c$, we get that $\gamma \circ \alpha$ is the identity map. This means that B_a and B_c are isomorphic. We conclude that contract Galois connections generated from Boolean algebra maps impose very rigid constraints on the Boolean algebras over which the contracts are defined.

Chapter 7

Syntax and the AG algebra

We have discussed the various algebraic operations one may carry on AG contracts. In this chapter we consider aspects of the implementation of these operations in a methodology. In particular, we carefully consider the role of the assumptions and guarantees in the design process.

7.1 The role of assumptions

Suppose we are asked to provide an implementation for a contract $\mathcal{C} = (A, G)$. We intuit that the more assumptions we make, the easier the design process should be. For example, if everything we are required to do is provided in the assumptions, the contract should tell that there is nothing left to do. In this case, we would have $A \leq G$, and $\mathcal{C}$ would be equivalent to the contract $(A, \mathcal{B})$. In other words, any component satisfies the guarantees, which means there is nothing left to implement. How can syntactic manipulations on contracts allow us to conclude this?

Contracts (A, G) and (A', G') are equivalent when

$$A = A' \text{ and } G \cup \neg A = G' \cup \neg A'.$$

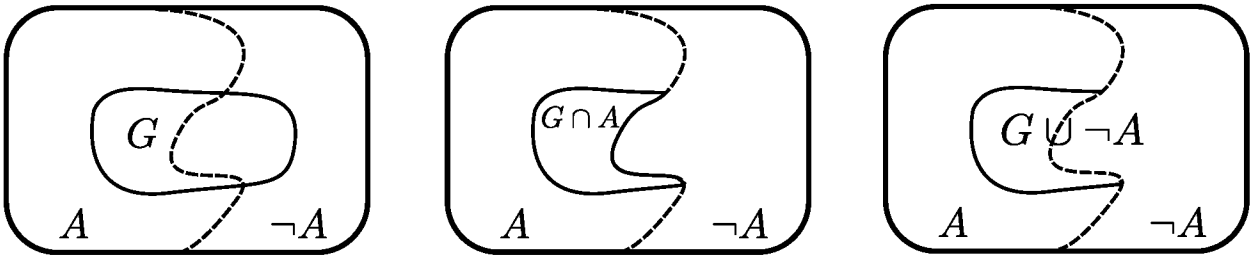


Figure 7.1: Given a contract $\mathcal{C} = (A, G)$, one may compute the contracts $(A, G \cap A)$ and $(A, G \cup \neg A)$, but these mappings are not invertible.

This means that a contract $\mathcal{C} = (A, G)$ is equivalent to $\mathcal{C}' = (A, G \cap A)$ and to $\mathcal{C}'' = (A, G \cup \neg A)$. Observe that $\mathcal{C}'$ and $\mathcal{C}''$ have the smallest and largest sets of guarantees among contracts in the same equivalence class. It is possible to compute $\mathcal{C}'$ and $\mathcal{C}''$ from $\mathcal{C}$, but we cannot go the other way around. Figure 7.1 illustrates this point. However, when we use syntactic expressions to represent assumptions and guarantees, the resulting formulas carry additional information which allow us to recover structure that would be lost by exclusively operating on sets.

Figure 7.2 shows a voltage amplifier with input i and output o. This component obeys the contract $\mathcal{C}$ with assumptions *the input voltage has magnitude smaller than 2* and guarantees *the output voltage is equal to the input*, i.e.,

$$\mathcal{C} = (|i| < 2, o = i)\,.$$

Observe that we are using syntactic expressions to denote assumptions and guarantees. For example, the assumptions of $\mathcal{C}$ are

$$A = \left\{(i, o) \in \mathbb{R}^2 \;\middle|\; |i| < 2\right\}.$$

Using syntax to denote properties offers the advantage that properties just need to refer to the variables on which they impose constraints. Note, for example, how the expression $|i| < 2$ used for the assumptions does not refer to the variable o, but its denotation is over all behaviors of systems variables, including o.

Now observe that the contract $\mathcal{C}$ is expressed in a way which is natural for a designer. Conder the equivalent contract $\mathcal{C}'$ with minimum guarantees:

$$\mathcal{C}' = (|i| < 2, (o = i) \wedge (|i| < 2)).$$

The assumptions are the same as before, but the guarantees now read, *the output voltage is equal to the input, and the input voltage has magnitude less than 2*. While this is logically correct, why should a device need to guarantee that "the input voltage has magnitude less than 2" when this is already provided by the assumptions of the contract? Similarly, the contract which is equivalent to $\mathcal{C}$ and which has the largest set of guarantees is

$$\mathcal{C}'' = (|i| < 2, (o = i) \vee \neg(|i| < 2)).$$

This contract has the same assumptions as $\mathcal{C}$, but the guarantees read, *the output voltage is equal to the input, or the input voltage has magnitude larger than or equal to 2*. Again, while this is logically correct, the implementation can assume it operates in a regime in which the input voltage is less than 2, so the second part we added to the original guarantees does not add information. Thus, in order to communicate specifications, we claim that the preferable contract is $\mathcal{C}$. In other words, a contract's guarantees should not impose constraints that the assumptions already impose. This does not yield the contract with smaller guarantees; it yields guarantees represented with fewer terms. Observe how we can simplify contract $\mathcal{C}'$:

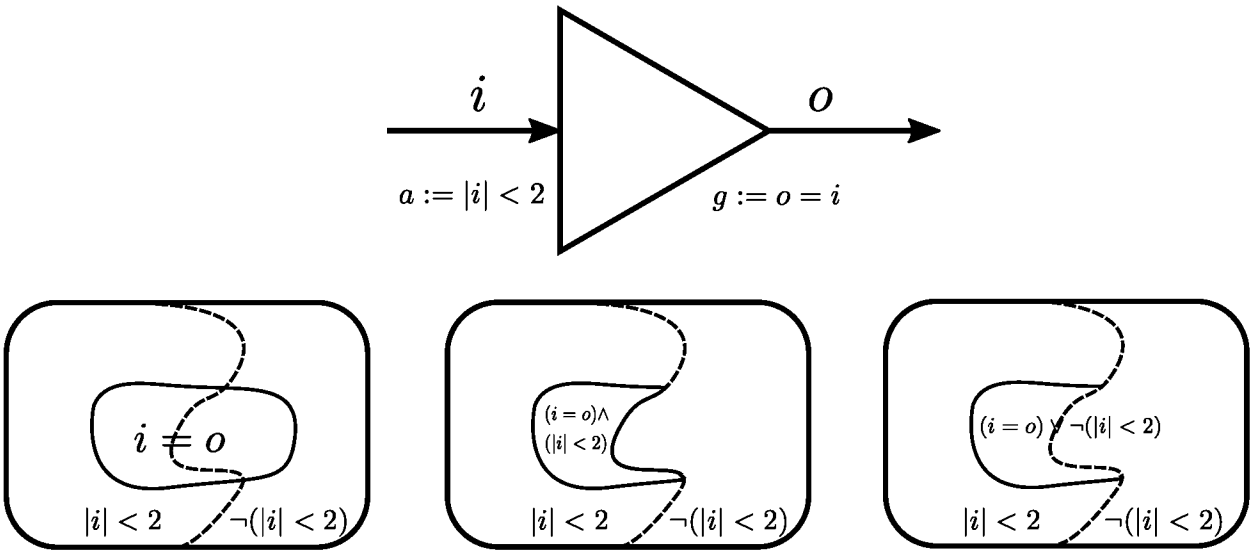


Figure 7.2: An amplifier with input i and output o obeying a contract $\mathcal{C}$

$$\begin{aligned}\mathcal{C}' &= (|i| < 2, (o = i) \wedge (|i| < 2)) \equiv (|i| < 2, (o = i) \wedge (|i| < 2) \vee \neg(|i| < 2)) \\ &= \underbrace{(|i| < 2, (o = i) \vee \neg(|i| < 2))}_{\mathcal{C}''} \equiv (|i| < 2, (o = i)) = \mathcal{C}.\end{aligned}$$

The first equivalence follows from contract saturation. The second equality is an application of the simplification rule

$$g \vee \neg a = g \wedge a \vee \neg a$$

from Boolean algebra. This rule is very useful to simplify contracts. It enables us to **remove from the guarantees any constraints imposed by the assumptions while maintaining logical equivalence**. The final equivalence is also obtained through saturation. We will record this result.

Proposition 7.1.1. *Let $\mathcal{C} = (A, G)$ be a contract and let $\mathfrak{a}$ and $\mathfrak{g}$ be sets of constraints such that $A = \bigwedge_{a \in \mathfrak{a}} a$ and $G = \bigwedge_{g \in \mathfrak{g}} g$. Let $G' = \bigwedge_{g \in \mathfrak{g} - \mathfrak{a}} g$. Then*

$$\mathcal{C} \equiv (A, G').$$

This example also shows that using syntax enabled us to recover G by having knowledge of $G \cap A$ and A, which is not possible to do by exclusively using the contract denotations, as illustrated in Figure 7.1.

7.2 Contracts in standard form

Proposition 7.1.1 applies to contracts whose assumptions and guarantees are given as lists of requirements. We claim that this is the most natural way to express assumptions and guarantees. It is common in industry to state a list of requirements that must hold in order for a device to deliver on its promises, also given as a list of constraints. In this section we will discuss the expression of contracts in this form.

7.2.1 Behaviors and terms

The most primitive element in system modeling is *the variable*. A variable is a named entity in our system having a progression of valuations. The notion of progression is given by a partial order, and valuations are provided by a topological space. We define a variable as the tuple

$$(V, \mathcal{X}_V, \mathcal{P}_V, \mathcal{B}_V),$$

where V is a name for the variable, $\mathcal{X}_V$ is a topological space where the variable takes values, and $\mathcal{P}_V$ is a partial order giving a notion of a progression. $\mathcal{B}_V$ is the *type* of the variable: a subset of $\mathcal{P}_V \to \mathcal{X}_V$, the set of functions from $\mathcal{P}_V$ to $\mathcal{X}_V$. We will use interchangeably the variable name and the tuple.

Example 7.2.1. *Suppose S is a variable that denotes the average temperature on the surface of the engine of a vehicle. We assume that temperature can be represented by any real number. Moreover, if we assume that S can take values for any real number, then we can set $\mathcal{B}_S = C(\mathbb{R}, \mathbb{R})$, i.e., any continuous function from the reals to the reals. Thus, the tuple representing this variable is*

$$(S, \mathbb{R}, \mathbb{R}, C(\mathbb{R}, \mathbb{R})).$$

Example 7.2.2. *Suppose A is a variable used to model the behavior of a label in a state machine. The valuations of A are Boolean, and A takes values over a countable sequence. Thus, this variable is given by*

$$(A, \{0, 1\}, \mathbb{N}, \mathbb{N} \to \{0, 1\}).$$

Example 7.2.3. *Suppose D is a variable used to model a constant in the system. The valuations of D are real. This variable is given by*

$$(D, \mathbb{R}, \{\bullet\}, \{\bullet\} \to \mathbb{R}).$$

For a set of variables **Vars**, the set of system behaviors is defined as

$$\mathcal{B} = \prod_{V \in \mathbf{Vars}} \mathcal{B}_V.$$

For the set of behaviors $\mathcal{B}$, the set of trace properties $2^{\mathcal{B}}$ is a Boolean algebra (i.e., it is closed under set intersection, union, and complementation). Having access to syntax means

having access to a set T of terms whose denotations are properties. We require the set of terms to be closed under the Boolean operations $\wedge$, $\vee$, and $\neg$. The denotation map must commute with these operations. Thus, there is a denotation map, $T \xrightarrow{\mathbf{Den}} 2^{\mathcal{B}}$, which respects Boolean algebra structure:

$$\begin{array}{ccc} T^{\times 2} & \xrightarrow{\wedge} & T \\ \downarrow{\scriptstyle \mathbf{Den}^{\times 2}} & & \downarrow{\scriptstyle \mathbf{Den}} \\ \left(2^{\mathcal{B}}\right)^{\times 2} & \xrightarrow{\cap} & 2^{\mathcal{B}} \end{array} \qquad \begin{array}{ccc} t, t' & \longmapsto & t \wedge t' \\ \downarrow & & \downarrow \\ \mathbf{Den}(t), \mathbf{Den}(t') & \longmapsto & \mathbf{Den}(t) \cap \mathbf{Den}(t') \end{array}$$

$$\begin{array}{ccc} T^{\times 2} & \xrightarrow{\vee} & T \\ \downarrow{\scriptstyle \mathbf{Den}^{\times 2}} & & \downarrow{\scriptstyle \mathbf{Den}} \\ \left(2^{\mathcal{B}}\right)^{\times 2} & \xrightarrow{\cup} & 2^{\mathcal{B}} \end{array} \qquad \begin{array}{ccc} t, t' & \longmapsto & t \vee t' \\ \downarrow & & \downarrow \\ \mathbf{Den}(t), \mathbf{Den}(t') & \longmapsto & \mathbf{Den}(t) \cup \mathbf{Den}(t') \end{array}$$

$$\begin{array}{ccc} T & \xrightarrow{\neg} & T \\ \downarrow{\scriptstyle \mathbf{Den}} & & \downarrow{\scriptstyle \mathbf{Den}} \\ 2^{\mathcal{B}} & \xrightarrow{\neg} & 2^{\mathcal{B}} \end{array} \qquad \begin{array}{ccc} t & \longmapsto & \neg t \\ \downarrow & & \downarrow \\ \mathbf{Den}(t) & \longmapsto & \neg\mathbf{Den}(t) \end{array}$$

We also assume there is a way to query the variables on which a term imposes constraints. We introduce a map $V : T \to 2^{\mathbf{Vars}}$ which returns a set of variables referenced in the term. This map is additive with respect to the term algebra:

$$\begin{array}{ccc} T^{\times 2} & \xrightarrow{\wedge} & T \\ \downarrow{\scriptstyle V^{\times 2}} & & \downarrow{\scriptstyle V} \\ \left(2^{\mathcal{B}}\right)^{\times 2} & \xrightarrow{\cup} & 2^{\mathcal{B}} \end{array} \qquad \begin{array}{ccc} t, t' & \longmapsto & t \wedge t' \\ \downarrow & & \downarrow \\ V(t), V(t') & \longmapsto & V(t) \cup V(t') \end{array}$$

$$\begin{array}{ccc} T^{\times 2} & \xrightarrow{\vee} & T \\ \downarrow{\scriptstyle V^{\times 2}} & & \downarrow{\scriptstyle V} \\ \left(2^{\mathcal{B}}\right)^{\times 2} & \xrightarrow{\cup} & 2^{\mathcal{B}} \end{array} \qquad \begin{array}{ccc} t, t' & \longmapsto & t \vee t' \\ \downarrow & & \downarrow \\ V(t), V(t') & \longmapsto & V(t) \cup V(t') \end{array}$$

$$\begin{array}{ccc} T & \xrightarrow{\neg} & T \\ \downarrow{\scriptstyle V} & & \downarrow{\scriptstyle V} \\ 2^{\mathcal{B}} & \xrightarrow{\mathrm{id}} & 2^{\mathcal{B}} \end{array} \qquad \begin{array}{ccc} t & \longmapsto & \neg t \\ \downarrow & & \downarrow \\ V(t) & \longmapsto & V(t) \end{array}$$

We will also assume the existence of a map $\mathbf{Vars}^{\times 2} \xrightarrow{r} \mathrm{Hom}(T, T)$ such that $v, v' \mapsto r_v^{v'}$, a map from T to T that commutes with the Boolean structure of T. We understand the map $r_v^{v'}$ as an operator that replaces references to v with references to v', provided the variables are of the same type. These maps have the following properties:

$$r_v^v = \mathrm{id},$$

i.e., renaming a variable with the same variable is the identity operator. We also have

$$r_{v''}^{v'''} r_v^{v'} = \begin{cases} r_v^{v'''} & v'' = v' \\ r_v^{v'} & v'' = v \\ r_v^{v'} r_{v''}^{v'''} & \text{otherwise} \end{cases}$$

Commutation with the term operations means that

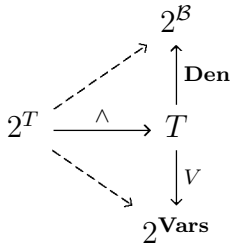
$$\begin{aligned} r_v^{v'}(t \wedge t') &= r_v^{v'} t \wedge r_v^{v'} t', \\ r_v^{v'}(t \vee t') &= r_v^{v'} t \vee r_v^{v'} t', \text{ and} \\ r_v^{v'}(\neg t) &= \neg(r_v^{v'} t). \end{aligned}$$

7.2.2 Standard form

A contract is defined as a pair of properties, $\mathcal{C} = (A, G)$, where $A, G \subseteq \mathcal{B}$. But we do not express properties using sets, but syntax. If we use terms from T in order to write properties, we can also write $\mathcal{C}$ directly using the terms of the syntax: $\mathcal{C} = (a, g)$, where $a, g \in T$ and $\mathbf{Den}(a) = A$ and $\mathbf{Den}(b) = B$. Finally, we consider a map, $2^T \xrightarrow{\wedge} T$, from sets of terms to behaviors, defined as

$$\mathfrak{t} \mapsto \bigwedge_{t \in \mathfrak{t}} t,$$

i.e., we take the conjunction of all terms in the set. This map allows us to extend the variable and denotations map to sets of terms:



Now we can think of assumptions and guarantees of contracts as sets of terms. We can thus write $\mathcal{C} = (\mathfrak{a}, \mathfrak{g})$, where $A = \bigwedge_{a \in \mathfrak{a}} \mathbf{Den}(a)$, and the same goes for the guarantees. We will say that contracts expressed in this way are contracts in *standard form*.

The operations we would like to compute on assume-guarantee contracts are summarized in Table 6.2. We will consider how composition and quotient affect contracts expressed in standard form. Take the case of the composition of contracts $\mathcal{C} = (a, g)$ and $\mathcal{C}' = (a', g')$ with their properties expressed with terms. The closed-form expression is

$$\mathcal{C} \parallel \mathcal{C}' = ((a \wedge a') \vee (a \wedge \neg g) \vee (a' \wedge \neg g'), (g \cap g') \vee (g \wedge \neg a') \vee (g' \wedge \neg a) \vee (\neg a \wedge \neg a')) \,.$$

We observe that both assumptions and guarantees are not expressed as conjunctions of terms, meaning that we would need to perform additional work to get the contracts in this form. The same happens with the other operations. However, we can make progress if we recall the meaning of composition.

Composition is an operation on contracts that provides the *smallest* specification observed by a system built out of components obeying their own contracts. This means that the system will also observe any specification which is looser than the contract provided by the composition operation. Thus, if the composition operation yields a contract with undesirable properties, we can seek abstractions which have the properties of interest. Similarly, quotient is the *largest* specification that we must compose with an existing specification so that the result meets a top-level spec. Thus, any specification smaller than the quotient will also complete our system. Merging and separation are, respectively, largest and smallest operations for other criteria.

So far we have argued that (i) the standard form of contracts has the desirable property that assumptions and guarantees are expressed as sets of terms or constraints, all of which must hold simultaneously, that (ii) standard contract operations yield results that break this standard form, and that (iii) perhaps there is additional processing we may do to the results of the operations in order to bring contracts to standard form. In the next section we see how these considerations apply to contract composition.

7.3 Computing the composition operation

Let's consider the situation described in Figure 7.3. There we have two amplifiers, M and M' connected in series. Suppose we are interested only in static behaviors. Then the universe of behaviors would be

$$\mathcal{B} = \mathbb{R}^3,$$

where the elements of $\mathbb{R}^3$ correspond to variables i, o, and o', respectively. These components obey the specifications $\mathcal{C}$ and $\mathcal{C}'$, respectively, where

$$\begin{aligned} \mathcal{C} &= (|i| < 2, o = i) \text{ and} \\ \mathcal{C}' &= (|o| < 1, o' = o)\,. \end{aligned} \tag{7.1}$$

We will explore how assumptions and guarantees can interact in order to yield useful abstractions and refinements of a contract.

Suppose we wish to provide a specification for the system represented by the series connection. Intuitively, we wish the specification to behave in such a way that the assumptions of both contracts are met. This would ensure that both components operate within their guarantees. Judging from the contracts we stated, we would like the system specification to be

$$\mathcal{C}^w = (|i| < 1, o' = i)\,,$$

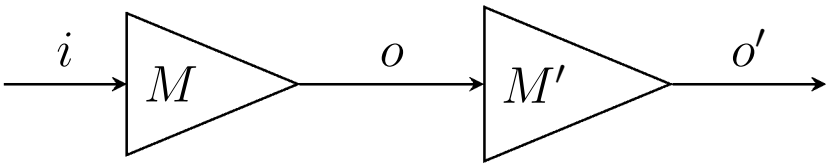


Figure 7.3: Two amplifiers, M and M', connected in series

as this contract says that, in order for both contract specifications to operate within their guarantees, the input needs to be restricted to $|i| < 1$. This is our wanted contract $\mathcal{C}^w$. This contract has the characteristic that assumptions are only given in terms of variables controlled by the environment, and guarantees are given in terms of variables controlled by the environment and by the component. No internal variables appear in assumptions or guarantees.

In order to find the specification of the system, the theory of contracts tells we should use the operation of composition. This operation yields the contract $\mathcal{C}^c = (a^c, g^c)$ (we will use lowercase to mean that assumptions and guarantees are given using syntactic expressions instead of denotations), where

$$\begin{aligned} a^c &= (|i| < 2 \wedge |o| < 1) \vee (|i| < 2 \wedge o \neq i) \vee (|o| < 1 \wedge o' \neq o) \text{ and} \\ g^c &= (|i| < 2 \rightarrow o = i) \wedge (|o| < 1 \rightarrow o' = o). \end{aligned}$$

This result poses two challenges. First, the assumptions include terms in which component specifications are violated, e.g., $|i| < 2 \wedge o \neq i$. If there is a behavior that meets this property, then this behavior won't be part of the intersection of the guarantees of the contracts being composed, i.e., it won't be part of $(|i| < 2 \rightarrow o = i)$. Second, even if the assumptions and guarantees of the contracts being composed are given in conjunctive normal form, the "logical or" in the assumptions of the contract breaks this form. How can we go from $\mathcal{C}^c$ to $\mathcal{C}^w$ through a sequence of abstractions?

Suppose we have two contracts $\mathcal{C}$ and $\mathcal{C}'$ whose assumptions and guarantees are given as

$$\begin{aligned} \mathcal{C} &= (a, g) \text{ and} \\ \mathcal{C}' &= (a', g'). \end{aligned}$$

We understand the assumptions and guarantees of these contracts as conjunctions of formulas, i.e., $a = \bigwedge_k r_k$, where the r_k are requirements. This matches how specifications are usually written. Also, the contracts $\mathcal{C}$ and $\mathcal{C}'$ just stated are not given in saturated form. Composing these contracts yields $\mathcal{C}^c = (a^c, g^c)$ with

$$\begin{aligned} a^c &= (a \wedge a') \vee (a \wedge \neg g) \vee (a' \wedge \neg g') \text{ and} \\ g^c &= (a \rightarrow g) \wedge (a' \rightarrow g') = g \wedge g' \vee \neg a^c. \end{aligned} \tag{7.2}$$

7.3.1 Refining the assumptions

We focus on the assumptions. As we said earlier, we wish the resulting system specification to operate in the regime in which the assumptions of both contracts are met. Thus, we refer to the term $a \wedge a'$ as the *stem* of the assumptions, the region where we want to operate. The terms $a \wedge \neg g$ and $a' \wedge \neg g'$ in the assumptions are not particularly useful as an end result of the system specification, as they represent situations in which components don't do what they are supposed to do when the assumptions were met. But these terms are not useless, either. We use these terms in order to rewrite the stem. The following result enables us to find refinements of the assumptions of the composition operation.

Proposition 7.3.1. *Let a_c be as in (7.2). Suppose there exist $\bar{a}$ such that $\bar{a} \wedge g' \leq a \wedge g'$. Then*

$$\bar{a} \wedge a' \leq a_c.$$

Moreover, if the inequality is satisfied as equality,

$$(\bar{a} \wedge a') \vee (a \wedge \neg g) \vee (a' \wedge \neg g') = a_c.$$

Proof.

$$\begin{aligned} a^c &= (a \wedge a') \vee (a \wedge \neg g) \vee (a' \wedge \neg g') \\ &= (a \wedge g' \wedge a') \vee (a \wedge \neg g) \vee (a' \wedge \neg g') \\ &\geq (\bar{a} \wedge g' \wedge a') \vee (a \wedge \neg g) \vee (a' \wedge \neg g') \\ &= (\bar{a} \wedge a') \vee (a \wedge \neg g) \vee (a' \wedge \neg g'). \end{aligned}$$

□

Proposition 7.3.1 allows us to find a refinement of the assumptions of the composition operation as a conjunction of terms. It also says that if we transform a into a' while maintaining $a \wedge g' = \bar{a} \wedge g'$, then the assumptions $\bar{a} \wedge a'$ are equivalent to a^c up to the unnecessary terms, i.e., the error terms $a \wedge \neg g$ and $a' \wedge \neg g'$. It is interesting to observe that Proposition 7.3.1 tells that we can use g' to transform a or g to transform a'. Can we do both transformations simultaneously? The following is a counterexample:

Example 7.3.2. *Let $g = g' = False$. We can set $\bar{a} = \bar{a}' = True$ since $False = \bar{a} \wedge g' \leq a \wedge g' = False$ and $False = \bar{a}' \wedge g \leq a' \wedge g = False$. In this case, we have $a^c = a \vee a'$, but $\bar{a} \wedge \bar{a}' = True$ fails to be a subset of a^c.*

7.3.2 Loosening the guarantees

Now that we have obtained a refinement of the assumptions, we pursue an abstraction of the guarantees.

Proposition 7.3.3. *Let $\mathcal{C}^c$ be as in (7.2) and $\bar{a}$ as in Proposition 7.3.1. Let $\bar{g}$ be such that $\bar{g} \wedge a' \wedge \bar{a} \geq g \wedge g' \wedge a' \wedge \bar{a}$. Then*

$$\bar{\mathcal{C}}^c = (\bar{a} \wedge a', \bar{g}).$$

is an abstraction of $\mathcal{C}^c$.

Proof. From Proposition 7.3.1, we know that $\bar{a} \wedge a' \leq a^c$. We also have

$$\begin{aligned} g^c &= (g \wedge g') \vee \neg a^c \\ &\leq (g \wedge g') \vee \neg(\bar{a} \wedge a') \\ &= (g \wedge g' \wedge a \wedge a') \vee \neg(\bar{a} \wedge a') \\ &\leq \bar{g} \vee \neg(\bar{a} \wedge a'). \end{aligned}$$

It follows that $(\bar{a} \wedge a', \bar{g}) \leq \mathcal{C}^c$. □

Algorithm 1 computes the composition of two assume-guarantee contracts given in standard form, providing a result in standard form. The correctness of its output follows from Propositions 7.3.3 and 7.3.1. We make the following observations:

- Lines 1 and 4 make use of Proposition 7.3.3 to refine the assumptions. As observed earlier, we can either use $\mathfrak{g}$ to simplify $\mathfrak{a}'$ or $\mathfrak{g}'$ to simplify $\mathfrak{a}$, but not both. The decision of which assumptions should be simplified (Lines 1 and 4) must be done using considerations *outside the algebra of AG contracts*. For example, the topology of the connection between the subsystems that implement the contracts may make the use of the guarantees of one of the contracts preferable. In Figure 7.3, one could argue that there is more benefit to be reaped by using the guarantees of M's contract to refine the assumptions of the contract $\mathcal{C}'$ of M' than by using the guarantees of $\mathcal{C}'$ to refine the assumptions of $\mathcal{C}$.

- The routines RefineWithSupport and AbstractWithSupport both take two lists of terms and generate a new list: the output $\bar{\mathfrak{a}} = \textsc{RefineWithSupport}(\mathfrak{a}, \mathfrak{g}')$ is such that
$$(\wedge\bar{\mathfrak{a}}) \wedge (\wedge\mathfrak{g}') \leq (\wedge\mathfrak{a}) \wedge (\wedge\mathfrak{g}'),$$
and the output $\bar{\mathfrak{g}} = \textsc{AbstractWithSupport}(\mathfrak{g}, \mathfrak{a}^c)$ is such that
$$(\wedge\bar{\mathfrak{g}}) \wedge (\wedge\mathfrak{a}^c) \geq (\wedge\mathfrak{g}) \wedge (\wedge\mathfrak{a}^c).$$

- These routines make use of the specific syntax of the constraints in order to carry out simplifications and abstractions of term lists.

7.3.3 Adding structure to contracts

Our discussion of Algorithm 1 revealed that the algebra of contracts does not contain enough information to make the decision needed in lines 1 and 4. We will introduce additional structure that will provide a selection criterion: the controllability of variables.

Many engineering components come with a notion of controllability of variables. For example, the component in Figure 7.2 is able to control variable o and unable to control i.

Algorithm 1 Overview of computation of the composition of AG contracts

Input: Contracts $\mathcal{C} = (\mathfrak{a}, \mathfrak{g})$ and $\mathcal{C}' = (\mathfrak{a}', \mathfrak{g}')$
Output: Abstraction of composition $(\mathfrak{a}^c, \mathfrak{g}^c)$

1: **if** Refining $\mathfrak{a}$ using $\mathfrak{g}'$ **then**
2: $\quad \bar{\mathfrak{a}} \leftarrow \textsc{RefineWithSupport}(\mathfrak{a}, \mathfrak{g}')$ $\quad \triangleright$ Find $\bar{\mathfrak{a}}$ such that $(\wedge\bar{\mathfrak{a}}) \wedge (\wedge\mathfrak{g}') \leq (\wedge\mathfrak{a}) \wedge (\wedge\mathfrak{g}')$
3: $\quad \mathfrak{a}^c \leftarrow \bar{\mathfrak{a}} \cup \mathfrak{a}'$
4: **else if** Refining $\mathfrak{a}'$ using $\mathfrak{g}$ **then**
5: $\quad \bar{\mathfrak{a}}' \leftarrow \textsc{RefineWithSupport}(\mathfrak{a}', \mathfrak{g})$ $\quad \triangleright$ Find $\bar{\mathfrak{a}}'$ such that $(\wedge\bar{\mathfrak{a}}') \wedge (\wedge\mathfrak{g}) \leq (\wedge\mathfrak{a}') \wedge (\wedge\mathfrak{g})$
6: $\quad \mathfrak{a}^c \leftarrow \bar{\mathfrak{a}}' \cup \mathfrak{a}$
7: **end if**
$\quad \triangleright$ Find $\mathfrak{g}^c$ such that $(\wedge\mathfrak{g}^c) \wedge (\wedge\mathfrak{a}^c) \geq (\wedge\mathfrak{g}) \wedge (\wedge\mathfrak{g}') \wedge (\wedge\mathfrak{a}^c)$
8: $\mathfrak{g}^c \leftarrow \textsc{AbstractWithSupport}(\mathfrak{g} \cup \mathfrak{g}', \mathfrak{a}^c)$
9: **return** $(\mathfrak{a}^c, \mathfrak{g}^c)$

For a specification of this component, it is not reasonable to make assumptions over o. The guarantees can impose constraints on the controlled variables when the uncontrolled variables meet certain conditions. In other words, the guarantees of a contract make prescriptive statements about controlled variables and descriptive statements of uncontrolled variables. We have the following definition:

Definition 7.3.4. *An IO contract is a tuple $(I, O, \mathcal{C})$, where $I, O \subseteq$ **Vars** are disjoint sets of input and output variables and $\mathcal{C} = (\mathfrak{a}, \mathfrak{g})$ is a contract in standard form such that $V(\mathfrak{a}) \subseteq I$ and $V(\mathfrak{g}) \subseteq I \cup O$.*

The composition of IO contracts is well-defined if the sets of output variables of the contracts being composed are disjoint. Algorithm 2 computes the composition of two IO contracts. Observe that the undefined lines 1 and 4 of Algorithm 1 are now replaced by checks that use the input and output variables of the contracts. If the outputs of $\mathcal{C}$ feed into the inputs of $\mathcal{C}'$, but not the opposite, then the guarantees of $\mathcal{C}$ are used to simplify the assumptions of $\mathcal{C}'$. Similarly, if the outputs of $\mathcal{C}'$ feed into the inputs of $\mathcal{C}$, but not the opposite, then the guarantees of $\mathcal{C}'$ are used to simplify the assumptions of $\mathcal{C}$. If both contracts have outputs that feed into the inputs of the other contract, we proceed as follows: we rename the outputs of $\mathcal{C}$ that feed into the inputs of $\mathcal{C}'$ to variables having names that haven't been used before. This renaming yields a contract $\bar{\mathcal{C}}$ having no outputs that feed into the inputs of $\mathcal{C}'$. We compose the contracts $\bar{\mathcal{C}}$ and $\mathcal{C}'$. Afterwards, we compose the result of this composition with a contract having true assumptions, and whose guarantees make the new variables equal to the variables they replaced. Figure 7.4 illustrates this process.

The routines RefineWithSupport and AbstractWithSupport have been extended with a third argument, a set of variables. This argument is used to indicate to the routines the variables that they should keep when carrying out simplifications—it's okay to eliminate all

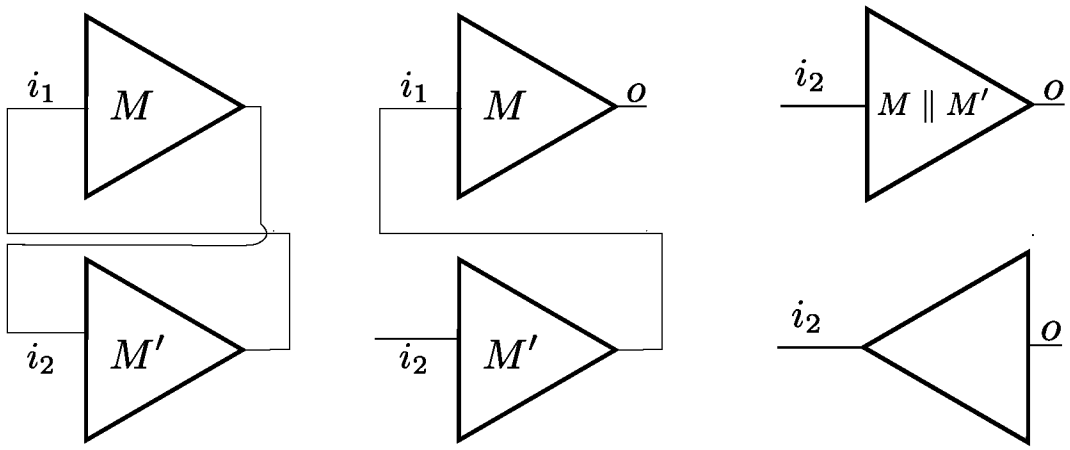


Figure 7.4: Two Amplifiers drive each others's inputs. To compose their contracts, we rename the output of the first amplifier o. This allows us to compose the contracts. Afterwards, we compose the composed contract with a contract that has True assumptions and whose guarantees make $i_1 = o$.

other variables. Observe that simplifications need an objective. We thus use as objectives the elimination of undesired variables and the reduction of the total number of terms. Usually, our interest is in eliminating from the assumptions internal and output variables, and in eliminating from the guarantees internal variables.

Algorithm 2 Computation of the composition of IO contracts

Input: IO contracts $(I, O, (\mathfrak{a}, \mathfrak{g}))$ and $(I', O', (\mathfrak{a}', \mathfrak{g}'))$
Output: Abstraction of composition $(\mathfrak{a}^c, \mathfrak{g}^c)$

$N \leftarrow (I \cap O') \cup (I' \cap O)$ ▷ Internal signals
$O^c \leftarrow (O \cup O') - N$
$I^c \leftarrow (I \cup I') - N$
if $O \cap O' \neq \emptyset$ **then**
  return Error: Contracts are not composable
else if $O' \cap I \neq \emptyset$ and $O \cap I' = \emptyset$ **then**
  ▷ Refining $\mathfrak{a}$ using $\mathfrak{g}'$
  $\bar{\mathfrak{a}} \leftarrow$ RefineWithSupport$(\mathfrak{a}, \mathfrak{g}', I^c)$ ▷ Find $\bar{\mathfrak{a}}$ s.t. $(\wedge\bar{\mathfrak{a}}) \wedge (\wedge\mathfrak{g}') \leq (\wedge\mathfrak{a}) \wedge (\wedge\mathfrak{g}')$
  $\mathfrak{a}^c \leftarrow \bar{\mathfrak{a}} \cup \mathfrak{a}'$
else if $O' \cap I = \emptyset$ and $O \cap I' \neq \emptyset$ **then**
  ▷ Refining $\mathfrak{a}'$ using $\mathfrak{g}$
  $\bar{\mathfrak{a}}' \leftarrow$ RefineWithSupport$(\mathfrak{a}', \mathfrak{g}, I^c)$ ▷ Find $\bar{\mathfrak{a}}'$ s.t. $(\wedge\bar{\mathfrak{a}}') \wedge (\wedge\mathfrak{g}) \leq (\wedge\mathfrak{a}') \wedge (\wedge\mathfrak{g})$
  $\mathfrak{a}^c \leftarrow \bar{\mathfrak{a}}' \cup \mathfrak{a}$
else if $O' \cap I = \emptyset$ and $O \cap I' = \emptyset$ **then**

$\mathfrak{a}^c \leftarrow \mathfrak{a} \cup \mathfrak{a}'$
else if $O' \cap I \neq \emptyset$ and $O \cap I' \neq \emptyset$ **then**
  ReplaceVars $\leftarrow O \cap I'$
  NewVars $\leftarrow \{\}$
  OldToNew $\leftarrow \{\}$
  for $v \in$ ReplaceVars **do**
    $v' \leftarrow$ NewVariableOfType(v)
    $\mathfrak{g} \leftarrow r_v^{v'}(\mathfrak{g})$
    NewVars $\leftarrow$ NewVars $\cup \{v'\}$
    OldToNew$[v] \leftarrow v'$
  end for
  $O \leftarrow (O -$ ReplaceVars$) \cup$ NewVars
  $(\bar{I}, \bar{O}, (\bar{\mathfrak{a}}, \bar{\mathfrak{g}})) \leftarrow$ ContractCompose$((I, O, (\mathfrak{a}, \mathfrak{g})), (I', O', (\mathfrak{a}', \mathfrak{g}')))$
  for $v \in$ ReplaceVars **do**
    $v' \leftarrow$ OldToNew$[v]$
    $\bar{\mathfrak{g}} \leftarrow r_{v'}^{v}(\bar{\mathfrak{g}})$
    $\bar{\mathfrak{a}} \leftarrow r_{v'}^{v}(\bar{\mathfrak{a}})$
  end for
  $\mathfrak{a}^c \leftarrow$ RefineWithSupport$(\bar{\mathfrak{a}}, \bar{\mathfrak{g}}, I^c)$
  $\mathfrak{g}^c \leftarrow$ AbstractWithSupport$(\bar{\mathfrak{g}}, \bar{\mathfrak{a}}^c, I^c \cup O^c)$
  return $(I^c, O^c, (\mathfrak{a}^c, \mathfrak{g}^c))$
end if
▷ Find $\mathfrak{g}^c$ such that $(\wedge\mathfrak{g}^c) \wedge (\wedge\mathfrak{a}^c) \geq (\wedge\mathfrak{g}) \wedge (\wedge\mathfrak{g}') \wedge (\wedge\mathfrak{a}^c)$
$\mathfrak{g}^c \leftarrow$ AbstractWithSupport$(\mathfrak{g} \cup \mathfrak{g}', \mathfrak{a}^c, I^c \cup O^c)$
return $(I^c, O^c, (\mathfrak{a}^c, \mathfrak{g}^c))$

7.4 Computing the quotient

Now we apply similar considerations to the computation of the quotient. Suppose we have two contacts $\mathcal{C} = (a, g)$ and $\mathcal{C}' = (a', g')$. The objective is to compute $\mathcal{C}/\mathcal{C}' = (a_q, g_q)$. The expressions are

$$a_q = a \wedge (\neg a' \vee g') \text{ and}$$
$$g_q = (a' \wedge g) \vee \neg a \vee (a' \wedge \neg g').$$

Proposition 7.4.1. *Suppose we have two contacts $\mathcal{C} = (a, g)$ and $\mathcal{C}' = (a', g')$. Let $\tilde{a} \geq a$, $\bar{g}$ be such that $g \wedge g' \geq \bar{g} \wedge g'$, and $\tilde{g}$ such that $a' \wedge \bar{g} \wedge a \geq \tilde{g} \wedge a$. Then $(\tilde{a}, \tilde{g}) \leq \mathcal{C}/\mathcal{C}'$.*

Proof. We have $\tilde{a} \geq a$. Moreover,

$$\begin{aligned} g_q &= (a' \wedge g) \vee \neg a \vee (a' \wedge \neg g') \\ &= (a' \wedge g \wedge g') \vee \neg a \vee (a' \wedge \neg g') \end{aligned}$$

$$
\begin{aligned}
&\geq (a' \wedge \bar{g} \wedge g') \vee \neg a \vee (a' \wedge \neg g') \\
&= (a' \wedge \bar{g} \wedge a) \vee \neg a \vee (a' \wedge \neg g') \\
&\geq (\tilde{g} \wedge a) \vee \neg a \vee (a' \wedge \neg g') \\
&= \tilde{g} \vee \neg a \vee (a' \wedge \neg g') \\
&\leq \tilde{g} \vee \neg \tilde{a}.
\end{aligned}
$$

It follows that $(\tilde{a}, \tilde{g}) \leq \mathcal{C}/\mathcal{C}'$. □

Algorithm 3 Overview of computation of the quotient of AG contracts

Input: Contracts $\mathcal{C} = (\mathfrak{a}, \mathfrak{g})$ and $\mathcal{C}' = (\mathfrak{a}', \mathfrak{g}')$
Output: Refinement of the quotient $(\mathfrak{a}^q \mathfrak{g}^q)$

1: $\mathfrak{a}^q \leftarrow$ ABSTRACT($\mathfrak{a}$)
2: $\bar{\mathfrak{g}} \leftarrow$ REFINEWITHSUPPORT($\mathfrak{g}, \mathfrak{g}'$)
3: $\mathfrak{g}^q \leftarrow$ REFINEWITHSUPPORT($\mathfrak{a}' \cup \bar{\mathfrak{g}}, \mathfrak{a}$)
4: **return** $(\mathfrak{a}^q, \mathfrak{g}^q)$

This proposition provides a way of obtaining a refinement of the quotient whose assumptions and guarantees are expressed in standard form. Algorithm 3 shows this procedure explicitly. Not reflected in the algorithm are mechanisms for carrying out the abstractions and refinements of specifications. As discussed when we considered the composition operation, additional data is needed to guide these routines. Possible goals of simplifications are elimination of variables in the assumptions or guarantees and reductions in the number of terms.

Now we consider the quotient using IO contracts. The outputs of $\mathcal{C}'$ must be disjoint from the inputs of $\mathcal{C}$, as the inputs of $\mathcal{C}$ are controlled by the environment. The outputs of the quotient must be either (i) the outputs of the top level contract that $\mathcal{C}'$ does not generate or (ii) the inputs to $\mathcal{C}'$ which are not inputs to $\mathcal{C}$. Similarly, the inputs of the quotient can be (i) the inputs of $\mathcal{C}$ or (ii) the outputs of $\mathcal{C}'$ which are not outputs of $\mathcal{C}$. These considerations are observed in Algorithm 4.

7.5 Constraints as partial orders

In this section we dedicate further attention to the structure of the terms in which our constraints are written. The set of terms T is a freely-generated Boolean algebra over a set of predicates, i.e.,

$$\langle \text{term} \rangle \quad \models \quad \langle \text{pred} \rangle \;\mid\; \langle \text{term} \rangle \wedge \langle \text{term} \rangle \;\mid\; \langle \text{term} \rangle \vee \langle \text{term} \rangle \;\mid\; \neg \langle \text{term} \rangle$$

Algorithm 4 Computation of the quotient of IO contracts

Input: Contracts $(I, O, (\mathfrak{a}, \mathfrak{g}))$ and $(I', O', (\mathfrak{a}', \mathfrak{g}'))$
Output: Refinement of the quotient $(\mathfrak{a}^q \mathfrak{g}^q)$

if $I \cap O' \neq \emptyset$ **then**
  return Error: The quotient is not defined
end if
$O^q = (O - O') \cup (I' - I)$
$I^q = (O' - O) \cup (I)$
$\mathfrak{a}^q \leftarrow \textsc{Abstract}(\mathfrak{a}, I^q)$
$\bar{\mathfrak{g}} \leftarrow \textsc{RefineWithSupport}(\mathfrak{g}, \mathfrak{g}', I^q \cup O^q)$
$\mathfrak{g}^q \leftarrow \textsc{RefineWithSupport}(\mathfrak{a}' \cup \bar{\mathfrak{g}}, \mathfrak{a}, I^q \cup O^q)$
return $(\mathfrak{a}^q, \mathfrak{g}^q)$

The predicates are statements that can be judged to be true or false. We will consider predicates having the syntax of a partial order:

$$\langle \text{pred} \rangle \quad ::= \quad \langle \text{expr} \rangle = \langle \text{expr} \rangle \;\mid\; \langle \text{expr} \rangle \leq \langle \text{expr} \rangle$$

Observe that predicates assert relations between expressions. Our expressions will be either variables or real numbers:

$$\langle \text{expr} \rangle \quad ::= \quad \langle \text{var} \rangle \;\mid\; \langle \text{real} \rangle$$

Having access to this syntax, we explore how the routines RefineWithSupport and AbstractWithSupport can be defined.

Algorithm 5 RefineWithSupport

Input: Set of terms $\mathfrak{c}$ to be refined, set of terms $\mathfrak{s}$ to aid the refinement, and set K of variables to keep
Output: Set of terms $\bar{\mathfrak{c}}$

▷ Build a graph whose vertices are expressions and whose edges exist
▷ if there is a term that makes the two expressions equal
procedure GetEqualityGraph($\mathfrak{t}$) **do**
  $R, E \leftarrow \emptyset$
  for $t \in \mathfrak{t}$ **do**
    if TermType(t) = Pred **and** PredType(t) = Equality **then**
      $E \leftarrow E \cup \{V(t)\}$

```
6:              R ← R ∪ V(t)
7:              return (R, E)
8:          end if
9:      end for
10: end procedure
11: c̄ ← c
    ▷ Replace variables with those we want to keep
12: (R, E) ← GetEqualityGraph(c̄)
13: for c ∈ ConnectedComponents(R, E) do
14:     if c ∩ K ≠ ∅ then
15:         v′ ← Choose(c ∩ K)
16:     else
17:         v′ ← Choose(c)
18:     end if
19:     for v ∈ c − K do
20:         c̄ ← r_v^{v′} c̄
21:     end for
22: end for
23: (R, E) ← GetEqualityGraph(s)
24: for c ∈ ConnectedComponents(R, E) do
25:     if c ∩ K ≠ ∅ then
26:         v′ ← Choose(c ∩ K)
27:     else
28:         v′ ← Choose(c)
29:     end if
30:     for v ∈ c − K do
31:         c̄ ← r_v^{v′} c̄
32:         s ← r_v^{v′} s
33:     end for
34: end for
35: return c̄
```

Algorithms 5 and 6 show the routines that carry out refinements and abstractions. The refinement routine looks for ways of renaming variables that can be eliminated with variables that can be kept. The abstraction routine eliminates terms with forbidden variables after having carried out a renaming operation. A further simplification we can carry out pertains the transitivity of the $\leq$ relation.

Our definition of the syntax for terms supports general constraints. Our definition of expressions is limited, supporting variable names and constants. By expanding the expressions to, e.g., linear expressions, we could increase the types of specifications to which these techniques can be applied. Similarly, by adding temporal operators to predicates, we could make statements over progressions of valuations.

Algorithm 6 AbstractWithSupport

Input: Set of terms $\mathfrak{c}$ to be refined, set of terms $\mathfrak{s}$ to aid the refinement, and set K of variables to keep
Output: Set of terms $\bar{\mathfrak{c}}$

1: $\bar{\mathfrak{c}} \leftarrow$ RefineWithSupport$(\mathfrak{c}, \mathfrak{s})$
2: **for** $t \in \bar{c}set$ **do**
3: **if** $V(t) \cap \neg K \neq \emptyset$ **then**
4: $\bar{\mathfrak{c}} \leftarrow -\{t\}$
5: **end if**
6: **end for**
7: **return** $\mathfrak{c}$

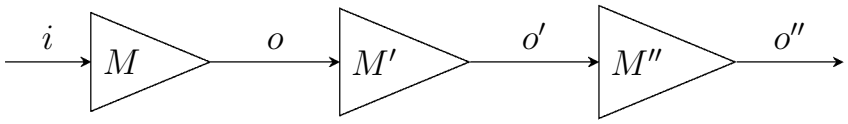


Figure 7.5: Sample system

7.5.1 Application

Consider the block diagram with three components shown in Figure 7.5. The contracts obeyed by the three components are

$$\mathcal{C} = (i < 2, o = i),$$
$$\mathcal{C}' = (o < 1, o' = o), \text{ and}$$
$$\mathcal{C}'' = (o' < 0, o'' = o).$$

We implemented simplification and contract operations, as described above. Our program receives the following input in order to represent the contracts just described:

```
iVar = Var("i")
oVar = Var("o")
assumptions = TermList({Term.LT(iVar, 2)})
guarantees = TermList({Term.EQ(oVar, iVar)})
cont = IoContract(assumptions, guarantees, {iVar}, {oVar})

opVar = Var("o'")
assumptions = TermList({Term.LT(oVar, 1)})
guarantees = TermList({Term.EQ(opVar, oVar)})
contp = IoContract(assumptions, guarantees, {oVar}, {opVar})
```

```
oppVar = Var("o''")
assumptions = TermList({Term.LT(opVar, 0)})
guarantees = TermList({Term.EQ(oppVar, opVar)})
contpp = IoContract(assumptions, guarantees, {opVar}, {oppVar})

print("Their composition is")
print(contp.compose(cont.compose(contpp)))
```

Executing the program yields the composition

```
A: (i < 0)
G: (o'' = i),
```

which represents our desired outcome of the composition operation.

Chapter 8

Hypercontracts

Assume-guarantee contracts, as presented in Chapter 2, are pairs of assumptions and guarantees expressed as trace properties. There are many important system attributes that cannot be modeled with trace properties. This chapter introduces *hypercontracts*, an assume guarantee theory which can support the expression of hyperproperties and in which we are able to instantiate several existing contract theories.

8.1 Introduction

The need for compositional algebraic frameworks to design and analyze reactive systems is widely accepted. The aim is to support distributed and decentralized system design based on a proper definition of *interfaces* supporting the specification of subsystems having a partially specified context of operation, and subsequently guaranteeing safe system integration. Over the last few decades, we have seen the introduction of several formalisms to do this: interface automata [28,54,55,62,135], process spaces [154], modal interfaces [16,118–120,169], assume-guarantee (AG) contracts [18], rely-guarantee reasoning [47, 83, 100, 101], and variants of these.

Since many contract frameworks have been proposed, there have also been efforts to systematize this knowledge by building high-level theories of which existing contract theories are instantiations. Bauer et al. [15] describe how to build a contract theory if one has a specification theory available. Benveniste et al. [19] provide a meta-theory that builds contracts starting from an algebra of components. They provide several operations on contracts and show how this meta-theory can describe, among others, interface automata, assume-guarantee contracts, modal interfaces, and rely-guarantee reasoning. This meta-theory is, however, low-level, specifying contracts as unstructured sets of environments and implementations. As a consequence, important concepts such as parallel composition and quotient of contracts are expressed in terms that are considered too abstract—see [19], chapter 4. For example, no closed form formula is given for the quotient besides its abstract definition as adjoint of parallel composition. This chapter introduces a theory, called *hypercontracts*, that

will address these drawbacks.

Among the various contract theories proposed so far, assume-guarantee (AG) contracts [18] require users to state the assumptions and guarantees of the specification explicitly, while interface theories express a specification as a game played between the specification environments and implementations. Experience tells that engineers in industry find the explicit expression of a contract's assumptions and guarantees natural (see [19] chapter 12), while interface theories are perceived as a less intuitive mechanism for writing specifications; however, interface theories in general come with the most efficient algorithms, making them excellent candidates for internal representations of specifications. Some authors ([19] chapter 10) have therefore proposed to translate contracts expressed as pairs (assumptions, guarantees) into some interface model, where algorithms are applied. This approach has the drawback that results cannot be traced back to the original (assumptions, guarantees) formulation.

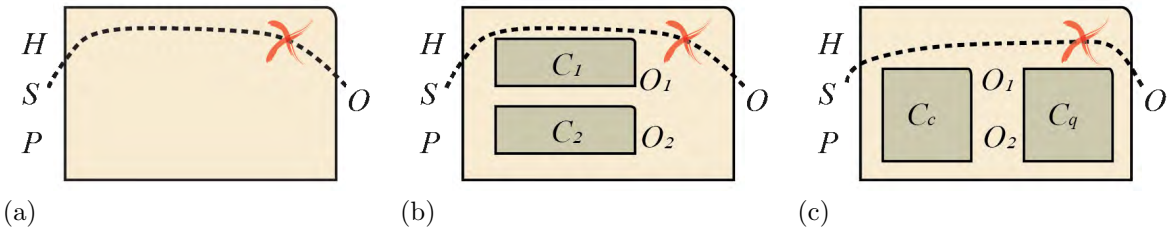
One of the difficulties with AG contracts is they only support environments and implementations that can be expressed using trace properties. While many attributes of interest can be expressed using trace properties, there are many important system attributes, such as non-interference, that need hyperproperties to be expressed. Hypercontracts allow environments and implementations to be expressed using arbitrary hyperproperties [45].

Hyperproperties are subsets of $2^{\mathcal{B}}$. Recall that each element of $2^{\mathcal{B}}$ defines a semantically-unique component. Thus, a component M satisfies a hyperproperty H if $M \in H$. One of our key contributions is an assume-guarantee theory that supports the expression of arbitrary hyperproperties. As we present our theory, we will use the following running example.

Example 8.1.1 (Running example)**.** *Consider the digital system shown in Figure 8.1a; this system is similar to those presented in [146, 167] to illustrate the non-interference property in security. Here, we have an s-bit secret data input S and an n-bit public input P. The system has an output O. There is also an input H that is equal to zero when the system is being accessed by a user with low-privileges, i.e., a user not allowed to use the secret data, and equal to one otherwise. We wish the overall system to satisfy the property that for all environments with $H = 0$, the implementations can only make the output O depend on P, the public data, not on the secret input S.*

A prerequisite for writing this requirement is to be able to express the property that "the output O depends on P, the public data, not on the secret input S". We claim that this property cannot be captured by a trace property. To see this, suppose for simplicity that all variables are 1-bit-long. A trace property that aims at expressing the independence from the secret for $O = P$ is

$$P = \left\{ \begin{array}{l} (P = 1, S = 1, O = 1), \\ (P = 0, S = 1, O = 0), \\ (P = 1, S = 0, O = 1), \\ (P = 0, S = 0, O = 0) \end{array} \right\}.$$



(a) (b) (c)

Figure 8.1: (a) A digital system with a secret input S and a public input P. The overall system must meet the requirement that the secret input does not affect the value of the output O when the signal H is deasserted (this signal is asserted when a privileged user uses the system). Our agenda for this running example is the following: (b) we will start with two components C_1 and C_2 satisfying respective hypercontracts $\mathcal{C}_1$ and $\mathcal{C}_2$ characterizing information-flow properties of their own; (c) the composition of these two hypercontracts, $\mathcal{C}_c$, will be derived. Through the quotient hypercontract $\mathcal{C}_q$, we will discover the functionality that needs to be added in order for the design to meet the top-level information-flow specification $\mathcal{C}$.

A valid implementation $M \subseteq P$ is the following set of traces:

$$M = \left\{ \begin{array}{l} (P=1, S=1, O=1), \\ (P=0, S=0, O=0) \end{array} \right\}.$$

However, the component M leaks the value of S in its output. We conclude that the independence does not behave as a trace property, and therefore, neither does non-interference. To overcome this, simply list all the subsets of P that satisfy the independence requirement:

$$\left\{ \begin{array}{l} (P{=}1, S{=}1, O{=}1), \\ (P{=}0, S{=}1, O{=}0), \\ (P{=}1, S{=}0, O{=}1), \\ (P{=}0, S{=}0, O{=}0) \end{array} \right\}, \left\{ \begin{array}{l} (P{=}1, S{=}1, O{=}1), \\ (P{=}1, S{=}0, O{=}1) \end{array} \right\}, \left\{ \begin{array}{l} (P{=}0, S{=}1, O{=}0), \\ (P{=}0, S{=}0, O{=}0) \end{array} \right\}$$

This precisely defines a subset of $2^{\mathcal{B}}$, i.e., a hyperproperty.

In our development, we will use hypercontracts first to express this top-level, assume-guarantee requirement, and then to find a component that added to a partial implementation of the system results in a design that meets the top-level specification. □

Contributions. In this chapter, we provide a theory of contracts, called *hypercontracts*, which generalizes existing theories of contracts while treating assumptions and guarantees as first-class citizens. This new AG theory supports arbitrary structured hyperproperties, including, e.g., non-interference and robustness.

The theory of hypercontracts is built in three stages. We begin with a theory of components. Then we state what are the sets of components that our theory can express; we call such objects compsets—compsets boil down to hyperproperties in behavioral formalisms [146]. From these compsets, we build hypercontracts. We provide closed-form

expressions for hypercontract manipulations. Then we show how the hypercontract theory applies to two specific cases: downward-closed hypercontracts and interface hypercontracts (equivalent to interface automata). The main difference between hypercontracts and the meta-theory of contracts [19] is that hypercontracts are more structured: the meta-theory of contracts defines a theory of components, and uses these components in order to define contracts. Hypercontracts use the theory of components to define compsets, which are the types of properties that we are interested in representing in a specific theory. Hypercontracts are built out of compsets, not out of components.

To summarize, our contributions are the following: (*i*) a new model of *hypercontracts* possessing a richer algebra than the metatheory of [19] and capable of expressing any lattice of hyperproperties and (*ii*) a calculus of *conic hypercontracts* offering finite representations of downward-closed hypercontracts.

8.2 The theory of hypercontracts

Our objective is to develop a theory of assume-guarantee reasoning for any kind of attribute of reactive systems. We do this in three steps:

1. we consider components coming with notions of preorder (e.g., simulation) and parallel composition;
2. we introduce the notion of a *compset*, a variation of the notion of hyperproperty, equipped with substantial algebraic structure;
3. we build *hypercontracts* as pairs of compsets with additional structure specifying legal environments and implementations.

8.2.1 Components

In the theory of hypercontracts, the most primitive concept is the component. Let $(\mathbb{M}, \leq)$ be a preorder. The elements $M \in \mathbb{M}$ are called *components*. We say that M is a subcomponent of M' when $M \leq M'$. If we represented components as automata, the statement "is a subcomponent of" would correspond to "is simulated by."

There exists a partial binary operation, $\|\colon \mathbb{M}, \mathbb{M} \to \mathbb{M}$, monotonic in both arguments, called *composition*. If $M \parallel M'$ is not defined, we say that M and M' are *non-composable* (and *composable* otherwise). A component E is an environment for component M if E and M are composable. We assume that composition is associative and commutative.

Example 8.2.1 (running example, cont'd)**.** *In order to reason about possible decompositions of the system shown in Figure 8.1a, we introduce the internal variables O_1 and O_2, as shown in Figure 8.1b. They have lengths o_1 and o_2, respectively. The output O has length o. For simplicity, we will assume that the behaviors of the entire system are stateless. In that case, the underlying set of components $\mathbb{M}$ must contain at least the following components:*

- *For $i \in \{1,2\}$, components with inputs H, S, P, and output O_i, i.e.,*

$$\{(H,S,P,O_1,O_2,O) \mid \exists f \in (2^1 \times 2^s \times 2^n \to 2^{o_i}).\, O_i = f(H,S,P)\}\,.$$

- *Components with inputs H, S, P, O_1, O_2, and output O, i.e.,*

$$\{(H,S,P,O_1,O_2,O) \mid \exists f \in (2^1 \times 2^s \times 2^n \times 2^{o_1} \times 2^{o_2} \to 2^o).\, O = f(H,S,P,O_1,O_2)\}\,.$$

- *Any subset of these components, as these correspond to restricting inputs to subsets of their domains.*

In this theory of components, composition is carried out via set intersection. So for example, if for $i \in \{1,2\}$ we have functions $f_i \in (2^1 \times 2^s \times 2^n \to 2^{o_i})$ and components $M_i = \{(H,S,P,O_1,O_2,O) \mid O_i = f_i(H,S,P)\}$, the composition of these objects is

$$M_1 \parallel M_2 = \left\{ (H,S,P,O_1,O_2,O) \;\middle|\; \begin{array}{l} O_1 = f_1(H,S,P) \\ O_2 = f_2(H,S,P) \end{array} \right\},$$

which is the set intersection of the components's behaviors. □

8.2.2 Compsets

CmpSet is a lattice whose objects, called *compsets*, are sets of components. The order of **CmpSet** is set inclusion. In general, not every set of components is an object of **CmpSet**. Compsets boil down to hyperproperties when the underlying component theory represents components as sets of behaviors. Since we assume **CmpSet** is a lattice, the greatest lower bounds and least upper bounds of finite sets are defined. Observe, however, that although the partial order of **CmpSet** is given by subsetting, the meet and join of **CmpSet** are not necessarily intersection and union, respectively, as the union or intersection of any two elements are not necessarily elements of **CmpSet**.

CmpSet comes with a notion of satisfaction. Suppose $M \in \mathbb{M}$ and H is a compset. We say that M *satisfies* H or conforms to H, written $M \models H$, when $M \in H$. For compsets H, H', we say that H *refines* H', written $H \leq H'$, when $M \models H \Rightarrow M \models H'$, i.e., when $H \subseteq H'$.

Since we assume **CmpSet** is a lattice, the greatest lower bounds and least upper bounds of finite sets are defined. Observe, however, that although the partial order of **CmpSet** is given by subsetting, the meet and join of **CmpSet** are not necessarily intersection and union, respectively, as the union or intersection of any two elements are not necessarily elements of **CmpSet**.

Example 8.2.2 (Running example: security)**.** Non-interference*, introduced by Goguen and Meseguer [72], is a common information-flow attribute, a prototypical example of a design quality which trace properties are unable to capture [45]. It can be expressed with hyperproperties, and is in fact one reason behind their introduction.*

Suppose σ is one of the behaviors that our system can display, understood as the state of memory locations through time. Some of those memory locations we call privileged*, some* unprivileged*. Let $L_0(\sigma)$ and $L_f(\sigma)$ be the projections of the behavior σ to the unprivileged memory locations of the system, at time zero, and at the final time (when execution is done). We say that a component M meets the non-interference hyperproperty when*

$$\forall \sigma, \sigma' \in M.\ L_0(\sigma) = L_0(\sigma') \Rightarrow L_f(\sigma) = L_f(\sigma'),$$

i.e., if two traces begin with the unprivileged locations in the same state, the final state of the unprivileged locations matches.

*Non-interference is a downward-closed hyperproperty [146,167], and a 2-safety hyperproperty. Hyperproperties called k-*safety *are those for the refutation of which one must provide at least k traces. In our example, to refute the hyperproperty, it suffices to show two traces that share the same unprivileged initial state, but which differ in the unprivileged final state.*

Regarding the system shown in Figure 8.1a, we require the top level component to generate the output O independently from the secret input S. We build our theory of compsets by letting the set $2^{\mathbb{M}}$ be the set of elements of **CmpSet**. *This means that any set of components is a valid compset. The components meeting the top-level non-interference property are those belonging to the compset $\{(H, S, P, O_1, O_2, O) \mid \exists f \in (2^1 \times 2^n \to 2^o).\, O = f(H, P)\}$, i.e., those components for which H and P are sufficient to evaluate O. This corresponds exactly to those components that are insensitive to the secret input S. The join and meet of these compsets is given by set union and intersection, respectively.* □

8.2.2.1 Composition and quotient

Composition in **CmpSet** is element-wise:

$$H \parallel H' = \left\{ M \parallel M' \;\middle|\; \begin{array}{l} M \models H,\ M' \models H', \text{ and} \\ M \text{ and } M' \text{ are composable} \end{array} \right\}. \tag{8.1}$$

Composition is total and monotonic, i.e., if $H' \leq H''$, then $H \parallel H' \leq H \parallel H''$. It is also commutative and associative, by the commutativity and associativity, respectively, of component composition.

We assume the existence of a second (but partial) binary operation on the objects of **CmpSet**. This operation is the right adjoint of composition: for compsets H and H', the residual H/H' (also called *quotient*), is defined by the universal property of the quotient. From the definition of composition, we must have

$$H/H' = \{M \in \mathbb{M} \mid \{M\} \parallel H' \subseteq H\}. \tag{8.2}$$

The definition of quotient for compsets does not require a notion of quotient for components. However, when such a notion exists, and depending on the structure of **CmpSet**, it can be used to simplify the computation of (8.2).

8.2.2.2 Convexity, co-convexity, and flatness

A compset H is *convex* if $M, M' \models H \Rightarrow M \parallel M' \models H$. In other words, H is convex if $H \parallel H \leq H$. A compset H is *co-convex* if $H \leq H \parallel H$.

If a compset is both convex and co-convex, it is called *flat*. Flat compsets H are precisely those that satisfy $H = H \parallel H$. If all compsets are flat, composition in **CmpSet** is idempotent.

Proposition 8.2.3. *Convexity, co-convexity, and flatness are preserved under composition.*

Proof. Suppose H and H' are convex compsets. Then $(H \parallel H') \parallel (H \parallel H') = (H \parallel H) \parallel (H' \parallel H') \leq H \parallel H'$, so their composition is convex. If H and H' are co-convex compsets, $H \parallel H' \leq (H \parallel H) \parallel (H' \parallel H') = (H \parallel H') \parallel (H \parallel H')$, so their composition is co-convex. The preservation of flatness follows from the preservation of convexity and co-convexity. □

Proposition 8.2.4. *If component composition is idempotent, all elements of* **CmpSet** *are co-convex.*

Proof. Suppose H is a compset and $M \models H$. Since component composition is idempotent, $M = M \parallel M$, so $M \models H \parallel H$. □

8.2.2.3 Downward-closed compsets

The set of components was introduced with a partial order. We say that a compset H is *downward-closed* when $M' \leq M$ and $M \models H$ imply $M' \models H$, i.e., if a component satisfies a downward-closed compset, so does its subcomponent. Section 8.4.4 treats downward-closed compsets in detail.

8.2.3 Hypercontracts

A hypercontract is a specification for a design element that tells what is required from the design element when it operates in an environment that meets the expectations of the hypercontract. Several ways of specifying hypercontracts can be considered.

Hypercontracts as pairs (environments, closed-system specification). In this setting, a hypercontract is a pair of compsets:

$$\mathcal{C} = (\mathcal{E}, \mathcal{S}) = (\text{environments, closed-system specification}).$$

$\mathcal{E}$ states the environments in which the object being specified must adhere to the specification. $\mathcal{S}$ states the requirements that the design element must fulfill when operating in an environment which meets the expectations of the hypercontract. We say that a component E *is an environment of hypercontract* $\mathcal{C}$, written $E \models^E \mathcal{C}$, if $E \models \mathcal{E}$. We say that a component M *is an implementation of* $\mathcal{C}$, written $M \models^I \mathcal{C}$, when $M \parallel E \models \mathcal{S}$ for all $E \models \mathcal{E}$. We thus

define the set of implementations $\mathcal{I}$ of $\mathcal{C}$ as the compset containing all implementations, i.e., as the quotient:

$$\text{implementations} = \mathcal{I} = \mathcal{S}/\mathcal{E}.$$

A hypercontract with a nonempty set of environments is called *compatible*. If it has a nonempty set of implementations, it is called *consistent*. For $\mathcal{S}$ and $\mathcal{I}$ as above, the compset $\mathcal{E}'$ defined as $\mathcal{E}' = \mathcal{S}/\mathcal{I}$ contains all environments in which the implementations of $\mathcal{C}$ satisfy the specifications of the hypercontract. Thus, we say that a hypercontract is *saturated* if its environments compset is as large as possible in the sense that adding more environments to the hypercontract would reduce its implementations. This means that $\mathcal{C}$ satisfies the following fixpoint equation: $\mathcal{E} = \mathcal{S}/\mathcal{I} = \mathcal{S}/(\mathcal{S}/\mathcal{E})$.

At a first sight, this notion of saturation may seem to go against what for assume-guarantee contracts are called contracts in canonical or saturated form, as we make the definition based on the environments instead of on the implementations. However, the two definitions for AG contracts and hypercontracts agree. Indeed, for AG contracts, this notion means that the contract $\mathcal{C} = (A, G)$ satisfies $G = G \cup \neg A$. For this AG contract, we can form a hypercontract as follows: if we take the set of environments to be $\mathcal{E} = 2^A$ (i.e., all subsets of A) and the closed system specs to be $\mathcal{S} = 2^G$, we get a hypercontract whose set of implementations is $2^{G \cup \neg A}$, which means that the hypercontract $(2^A, 2^G)$ is saturated.

Hypercontracts as pairs (environments, implementations). Another way to interpret a hypercontract is by telling explicitly which environments and implementations it supports. Thus, we would write the hypercontract as $\mathcal{C} = (\mathcal{E}, \mathcal{I})$. Assume-guarantee theories can differ as to the most convenient representation for their hypercontracts. Moreover, some operations on hypercontracts find their most convenient expression in terms of implementations (e.g., parallel composition), and some in terms of the closed system specifications (e.g., strong merging).

The lattice **Contr** *of hypercontracts.* Just as with **CmpSet**, we define **Contr** as a lattice formed by putting together two compsets in one of the above two ways. Not every pair of compsets is necessarily a valid hypercontract. We will define soon the operations that give rise to this lattice.

8.2.3.1 Preorder

We define a preorder on hypercontracts as follows: we say that $\mathcal{C}$ *refines* $\mathcal{C}'$, written $\mathcal{C} \leq \mathcal{C}'$, when every environment of $\mathcal{C}'$ is an environment of $\mathcal{C}$, and every implementation of $\mathcal{C}$ is an implementation of $\mathcal{C}'$, i.e., $E \models^E \mathcal{C}' \Rightarrow E \models^E \mathcal{C}$ and $M \models^I \mathcal{C} \Rightarrow M \models^I \mathcal{C}'$. We can express this as

$$\mathcal{E}' \leq \mathcal{E} \text{ and } \mathcal{S}/\mathcal{E} = \mathcal{I} \leq \mathcal{I}' = \mathcal{S}'/\mathcal{E}'.$$

Any two $\mathcal{C}, \mathcal{C}'$ with $\mathcal{C} \leq \mathcal{C}'$ and $\mathcal{C}' \leq \mathcal{C}$ are said to be *equivalent* since they have the same environments and the same implementations. We now obtain some operations using preorders which are defined as the LUB or GLB of **Contr**. We point out that the expressions we obtain are unique up to the preorder, i.e., up to hypercontract equivalence.

8.2.3.2 GLB and LUB

From the preorder just defined, the GLB of $\mathcal{C}$ and $\mathcal{C}'$ satisfies: $M \models^I \mathcal{C} \wedge \mathcal{C}'$ if and only if $M \models^I \mathcal{C}$ and $M \models^I \mathcal{C}'$; and $E \models^E \mathcal{C} \wedge \mathcal{C}'$ if and only if $E \models^E \mathcal{C}$ or $E \models^E \mathcal{C}'$.

Conversely, the least upper bound satisfies $M \models^I \mathcal{C} \vee \mathcal{C}'$ if and only if $M \models^I \mathcal{C}$ or $M \models^I \mathcal{C}'$, and $E \models^E \mathcal{C} \vee \mathcal{C}'$ if and only if $E \models^E \mathcal{C}$ and $E \models^E \mathcal{C}'$.

The lattice **Contr** has hypercontracts for objects (up to contract equivalence), and meet and join as just described.

8.2.3.3 Parallel composition

The composition of hypercontracts $\mathcal{C}_i = (\mathcal{E}_i, \mathcal{I}_i)$ for $1 \leq i \leq n$, denoted $\|_i \mathcal{C}_i$, is the smallest hypercontract $\mathcal{C}' = (\mathcal{E}', \mathcal{I}')$ (up to equivalence) meeting the following requirements:

- any composition of implementations of all $\mathcal{C}_i$ is an implementation of $\mathcal{C}'$; and

- for any $1 \leq j \leq n$, any composition of an environment of $\mathcal{C}'$ with implementations of all $\mathcal{C}_i$ (for $i \neq j$) yields an environment for $\mathcal{C}_j$.

These requirements were stated for the first time by Abadi and Lamport [1]. Using our notation, this composition principle becomes

$$\begin{aligned} \mathcal{C} \parallel \mathcal{C}' &= \bigwedge \left\{ \begin{array}{l|l} (\mathcal{E}', \mathcal{I}') & \left[\begin{array}{l} \mathcal{I}_1 \parallel \ldots \parallel \mathcal{I}_n \leq \mathcal{I}', \text{ and} \\ \mathcal{E}' \parallel \mathcal{I}_1 \parallel \ldots \parallel \hat{\mathcal{I}}_j \parallel \ldots \parallel \mathcal{I}_n \leq \mathcal{E}_j \\ \text{for all } 1 \leq j \leq n \end{array} \right] \\ \in \mathbf{Contr} & \end{array} \right\} \\ &= \bigwedge \left\{ \begin{array}{l|l} (\mathcal{E}', \mathcal{I}') & \left[\begin{array}{l} \mathcal{I}_1 \parallel \ldots \parallel \mathcal{I}_n \leq \mathcal{I}', \text{ and} \\ \mathcal{E}' \leq \bigwedge_{1 \leq j \leq n} \frac{\mathcal{E}_j}{\mathcal{I}_1 \parallel \ldots \parallel \hat{\mathcal{I}}_j \parallel \ldots \parallel \mathcal{I}_n} \end{array} \right] \\ \in \mathbf{Contr} & \end{array} \right\}, \end{aligned} \tag{8.3}$$

where the notation $\hat{\mathcal{I}}_j$ indicates that the composition $\mathcal{I}_1 \parallel \ldots \parallel \hat{\mathcal{I}}_j \parallel \ldots \parallel \mathcal{I}_n$ includes all terms $\mathcal{I}_i$, except for $\mathcal{I}_j$.

Example 8.2.5 (Running example, parallel composition)**.** *Coming back to the example shown in Figure 8.1, we want to state a requirement for the top-level component that for all environments with $H = 0$, the implementations can only make the output O depend on P, the public data. We will write a hypercontract for the top-level. We let $\mathcal{C} = (\mathcal{E}, \mathcal{I})$, where*

$$\begin{aligned} \mathcal{E} &= \{M \in \mathbb{M} \mid \forall (H, S, P, O_1, O_2, O) \in M.\, H = 0\} \\ \mathcal{I} &= \{M \in \mathbb{M} \mid \exists f \in (2^n \to 2^o). \forall (H, S, P, O_1, O_2, O) \in M.\, H = 0 \to O = f(P)\}. \end{aligned}$$

The environments are all those components only defined for $H = 0$. The implementations are those such that the output is a function of P when $H = 0$.

Let $f^* : 2^n \to 2^o$. Suppose we have two hypercontracts that require their implementations to satisfy the function $O_i = f^*(P)$, one implements it when $S = 0$, and the other when $S \neq 0$. For simplicity of syntax, let s_1 and s_2 be the propositions $S = 0$ and $S \neq 0$, respectively.

Let the two hypercontracts be $\mathcal{C}_i = (\mathcal{E}_i, \mathcal{I}_i)$ for $i \in \{1, 2\}$. We won't place restrictions on the environments for these hypercontracts, so we obtain $\mathcal{E}_i = \mathbb{M}$ and

$$\mathcal{I}_i = \{M \in \mathbb{M} \mid \forall (H, S, P, O_1, O_2, O) \in M. s_i \to O_i = f^*(P)\}.$$

We now evaluate the composition of these two hypercontracts: $\mathcal{C}_c = \mathcal{C}_1 \parallel \mathcal{C}_2 = (\mathcal{E}_c, \mathcal{I}_c)$, yielding $\mathcal{E}_c = \mathbb{M}$ and

$$\begin{aligned}\mathcal{I}_c = \{&M \in \mathbb{M} \mid \forall (H, S, P, O_1, O_2, O) \in M. \\ &(s_1 \to O_1 = f^*(P)) \wedge (s_2 \to O_2 = f^*(P))\}.\end{aligned}$$

8.2.3.4 Mirror or reciprocal

We assume we have an additional operation on hypercontracts, called both mirror and reciprocal, which flips the environments and implementations of a hypercontract: $\mathcal{C}^{-1} = (\mathcal{E}, \mathcal{I})^{-1} = (\mathcal{I}, \mathcal{E})$ and $\mathcal{C}^{-1} = (\mathcal{E}, \mathcal{S})^{-1} = (\mathcal{S}/\mathcal{E}, \mathcal{S})$. This notion gives us, so to say, the hypercontract obeyed by the environment. The introduction of this operation assumes that for every hypercontract $\mathcal{C}$, its reciprocal is also an element of **Contr**. Moreover, we assume that, when the infimum of a collection of hypercontracts exists, the following identity holds:

$$\left(\bigwedge_i \mathcal{C}_i\right)^{-1} = \bigvee_i \mathcal{C}_i^{-1}. \tag{8.4}$$

8.2.3.5 Hypercontract quotient

The *quotient* or residual for hypercontracts $\mathcal{C} = (\mathcal{E}, \mathcal{I})$ and $\mathcal{C}'' = (\mathcal{E}'', \mathcal{I}'')$, written $\mathcal{C}''/\mathcal{C}$, has the universal property of the quotient, namely $\forall \mathcal{C}'.\ \mathcal{C} \parallel \mathcal{C}' \leq \mathcal{C}''$ if and only if $\mathcal{C}' \leq \mathcal{C}''/\mathcal{C}$. We can obtain a closed-form expression using the reciprocal:

Proposition 8.2.6. *The hypercontract quotient obeys $\mathcal{C}''/\mathcal{C} = ((\mathcal{C}'')^{-1} \parallel \mathcal{C})^{-1}$.*

Proof.

$$\begin{aligned}\mathcal{C}''/\mathcal{C} &= \bigvee \{\mathcal{C}' \mid \mathcal{C} \parallel \mathcal{C}' \leq \mathcal{C}''\} = \bigvee \left\{ (\mathcal{E}', \mathcal{I}') \;\middle|\; \left[\begin{array}{l} \mathcal{I} \parallel \mathcal{I}' \leq \mathcal{I}'', \\ \mathcal{E}'' \parallel \mathcal{I} \leq \mathcal{E}', \text{ and} \\ \mathcal{E}'' \parallel \mathcal{I}' \leq \mathcal{E} \end{array}\right] \right\} \\ &= \left(\left(\bigvee \left\{ (\mathcal{E}', \mathcal{I}') \;\middle|\; \left[\begin{array}{l} \mathcal{I} \parallel \mathcal{I}' \leq \mathcal{I}'', \\ \mathcal{E}'' \parallel \mathcal{I} \leq \mathcal{E}', \text{ and} \\ \mathcal{E}'' \parallel \mathcal{I}' \leq \mathcal{E} \end{array}\right] \right\} \right)^{-1} \right)^{-1} \\ &\overset{(8.4)}{=} \left(\bigwedge \left\{ (\mathcal{I}', \mathcal{E}') \;\middle|\; \left[\begin{array}{l} \mathcal{I} \parallel \mathcal{I}' \leq \mathcal{I}'', \\ \mathcal{E}'' \parallel \mathcal{I} \leq \mathcal{E}', \text{ and} \\ \mathcal{E}'' \parallel \mathcal{I}' \leq \mathcal{E} \end{array}\right] \right\} \right)^{-1} \\ &= \left(\bigwedge \left\{ (\mathcal{I}', \mathcal{E}') \;\middle|\; \left[\begin{array}{l} \mathcal{E}'' \parallel \mathcal{I} \leq \mathcal{E}', \\ \mathcal{I}' \parallel \mathcal{I} \leq \mathcal{I}'', \text{ and} \\ \mathcal{I}' \parallel \mathcal{E}'' \leq \mathcal{E} \end{array}\right] \right\} \right)^{-1}\end{aligned}$$

$$= \left((\mathcal{C}'')^{-1} \parallel \mathcal{C}\right)^{-1}.$$

□

Example 8.2.7 (Running example, quotient)**.** *We use the quotient to find the specification of the component that we need to add to the system shown in Figure 8.1c in order to meet the top level contract $\mathcal{C}$. To compute the quotient, we use* (8.8)*. We let $\mathcal{C}/\mathcal{C}_c = (\mathcal{E}_q,\ \mathcal{I}_q)$ and obtain $\mathcal{E}_q = \mathcal{E} \wedge \mathcal{I}_c$ and*

$$\begin{aligned}\mathcal{I}_q = &\{M \in \mathbb{M} \mid \exists f \in (2^n \to 2^o)\forall(H, S, P, O_1, O_2, O)\\ &\in M.\,((s_1 \to O_1{=}f^*(P)) \wedge (s_2 \to O_2{=}f^*(P))) \to (H = 0 \to O{=}f(P))\}.\end{aligned}$$

We can refine the quotient by lifting any restrictions on the environments, and picking from the implementations the term with $f = f^*$. Observe that f^* is a valid choice for f. This yields the hypercontract $\mathcal{C}_3 = (\mathcal{E}_3, \mathcal{I}_3)$, defined as $\mathcal{E}_3 = \mathbb{M}$ and

$$\begin{aligned}\mathcal{I}_3 = &\{M \in \mathbb{M} \mid \forall(H, S, P, O_1, O_2, O) \in M.\\ &((s_1 \to O_1{=}f^*(P)) \wedge (s_2 \to O_2{=}f^*(P))) \to O{=}f^*(P)\}.\end{aligned}$$

A further refinement of this hypercontract is $\mathcal{C}_r = (\mathcal{E}_r, \mathcal{I}_r)$, where $\mathcal{E}_r = \mathbb{M}$ and

$$\mathcal{I}_r = \{M \in \mathbb{M} \mid \forall(H, S, P, O_1, O_2, O) \in M.\,((s_1 \to O{=}O_1) \wedge (s_2 \to O{=}O_2))\}.$$

By the properties of the quotient, composing this hypercontract, which knows nothing about f^*, with $\mathcal{C}_c$ will yield a hypercontract which meets the non-interference hypercontract $\mathcal{C}$. Note that this hypercontract is consistent, i.e., it has implementations (in general, refining may lead to inconsistency). □

8.2.3.6 Merging

The composition of two hypercontracts yields the specification of a system comprised of two design objects, each adhering to one of the hypercontracts being composed. Another important operation on hypercontracts is viewpoint merging, or *merging* for short. It can be the case that the same design element is assigned multiple specifications corresponding to multiple viewpoints, or design concerns [18, 161] (e.g., functionality and a performance criterion). Suppose $\mathcal{C}_1 = (\mathcal{E}_1, \mathcal{S}_1)$ and $\mathcal{C}_2 = (\mathcal{E}_2, \mathcal{S}_2)$ are the hypercontracts we wish to merge. Two slightly different operations can be considered as candidates for formalizing viewpoint merging:

- A *weak merge* which is the GLB; and

- A *strong merge* which states that environments of the merger should be environments of both $\mathcal{C}_1$ and $\mathcal{C}_2$ and that the closed systems of the merger are closed systems of both $\mathcal{C}_1$ and $\mathcal{C}_2$. If we let $\mathcal{C}_1 \bullet \mathcal{C}_2 = (\mathcal{E}, \mathcal{I})$, we have

$$\mathcal{E} = \vee\{\mathcal{E}' \in \mathbf{CmpSet} \mid \mathcal{E}' \leq \mathcal{E}_1 \wedge \mathcal{E}_2 \text{ and } \exists \mathcal{C}'' = (\mathcal{E}'', \mathcal{I}'') \in \mathbf{Contr}.\ \mathcal{E}' = \mathcal{E}''\}$$
$$\mathcal{I} = \vee \left\{ \mathcal{I}' \in \mathbf{CmpSet} \;\middle|\; \begin{array}{l} \mathcal{I}' \leq (\mathcal{S}_1 \wedge \mathcal{S}_2)/\mathcal{E} \text{ and} \\ (\mathcal{E}, \mathcal{I}) \in \mathbf{Contr} \end{array} \right\}.$$

The difference is that, whereas the commitment to satisfy $\mathcal{S}_2$ survives under the weak merge when the environment fails to satisfy $\mathcal{E}_1$, no obligation survives under the strong merge. This distinction was proposed in [177] under the name of weak/strong assumptions.

8.2.4 An example on robustness

Now we explore assume-guarantee specifications of autonomous vehicles. We will deal with their safety and the robustness of their perception components. In order to consider the perception components, we will build our model using a pair (X, O), where $X \in S$ is the input image, belonging to a set S of images, and $O \in CS$ is the classification of the image X, an element of the classification space CS. To deal with safety, we will consider pairs $(v, \Delta s)$, where v represents the state of the vehicle with domain SP, and Δs is the maximum amount of time that it takes the vehicle to come to a full stop. Thus, every component $M \in \mathbb{M}$ is of the form

$$M = \left\{(X, O, v, \Delta_s) \in S \times CS \times SP \times \mathbb{R}^+ \;\middle|\; \exists f \in S \to CS.\ O = f(X)\right\}.$$

As discussed in Seshia et al. [183], certain robustness properties of data-driven components are hyperproperties. Robustness properties usually take the form $d(x, y) < \delta \Rightarrow D(f(x), f(y)) < \varepsilon$, where d and D are distance functions. The property says that points that are close should have similar classifications. As two points are needed to provide evidence that a function is not robust, these are 2-safety hyperproperties. We will state a specification for our vehicles that requires their perception components to be robust. Suppose the input space S is partitioned in sets S_i. We want our vehicle to meet the following top-level specification:

$$\mathcal{C} = \left(\mathbb{M}, \left\{ M \in \mathbb{M} \;\middle|\; \begin{array}{l} \forall (x_k, o_k, v_k, \Delta s_k), (x_l, o_l, v_l, \Delta s_l) \in M. \\ \bigwedge_i x_k, x_l \in S_i \to |o_k - o_l| \leq \varepsilon \end{array} \right\} \right).$$

Suppose our vehicle obeys the specification $\mathcal{C}_a$ given by

$$\mathcal{C}_a = \left(\mathbb{M}, \left\{ M \in \mathbb{M} \;\middle|\; \begin{array}{l} \forall (x_k, o_k, v_k, \Delta s_k), (x_l, o_l, v_l, \Delta s_l) \in M. \\ \bigwedge_i x_k, x_l \in S_i \to |o_k - o_l| \leq \varepsilon_i \end{array} \right\} \right).$$

This specification says that the perception component in each region S_i should have a robustness ε_i. Suppose that there is a $j \in \mathbb{N}$ such that $\varepsilon_i \leq \varepsilon$ for all $i \leq j$ and $\varepsilon_i > \varepsilon$ otherwise.

The contract quotient is $\mathcal{C}_q = (\mathcal{E}_q, \mathcal{I}_q)$, where $\mathcal{E}_q = \mathcal{I}_a$ and

$$\mathcal{I}_q = \frac{\left\{ M \in \mathbb{M} \;\middle|\; \begin{array}{l} \forall (x_k, o_k, v_k, \Delta s_k), (x_l, o_l, v_l, \Delta s_l) \in M. \\ \bigwedge_i x_k, x_l \in S_i \rightarrow |o_k - o_l| \leq \varepsilon \end{array} \right\}}{\left\{ M \in \mathbb{M} \;\middle|\; \begin{array}{l} \forall (x_k, o_k, v_k, \Delta s_k), (x_l, o_l, v_l, \Delta s_l) \in M. \\ \bigwedge_i x_k, x_l \in S_i \rightarrow |o_k - o_l| \leq \varepsilon_i \end{array} \right\}},$$

where we used the horizontal bar to denote the compset quotient. By the definition of the contract quotient, any refinement of $\mathcal{C}_q$ is a solution to our problem, namely, what is the specification that we have to compose with a specification $\mathcal{C}_a$ in order for the result to meet a goal specification $\mathcal{C}$. We thus compute a refinement of the quotient that we just obtained:

$$\mathcal{C}_b = \left(\mathbb{M}, \left\{ M \in \mathbb{M} \;\middle|\; \begin{array}{l} \forall (x_k, o_k, v_k, \Delta s_k), (x_l, o_l, v_l, \Delta s_l) \in M. \\ \bigwedge_{i>j} x_k, x_l \in S_i \rightarrow |o_k - o_l| \leq \varepsilon \end{array} \right\} \right).$$

Observe how using the quotient we were able to obtain a specification $\mathcal{C}_b$ that contains exactly what needs to be fixed in the component adhering to hypercontract $\mathcal{C}_a$ in order for it to meet the top-level specification $\mathcal{C}$. Moreover, the specification $\mathcal{C}_b$ does not contain any information about $\mathcal{C}_a$.

One of the uses of hypercontracts is in handling multiple viewpoints. Suppose that the robust perception specification is given to a vehicle on top of other specifications, such as safety. For example, suppose there is a specification that says that if the state of the vehicle v is inside a safety set T, then the amount of time Δs that it takes the vehicle to come to a full stop is a most P. We can write the spec

$$\mathcal{C}_s = (v \in T, \Delta s < P).$$

By using strong merging, we can get into a single top-level hypercontract the specification of the perception and the safety viewpoints, as follows:

$$\left(v \in T, \left\{ M \in \mathbb{M} \;\middle|\; \begin{array}{l} \forall (x_k, o_k, v_k, \Delta s_k), (x_l, o_l, v_l, \Delta s_l) \in M. \\ \Delta s_k, \Delta s_l < P \wedge \bigwedge_i x_k, x_l \in S_i \rightarrow |o_k - o_l| \leq \varepsilon \end{array} \right\} \right).$$

This specification summarizes the perception and safety viewpoints of the vehicle. As robustness is a hyperproperty, we cannot use AG contracts to reason about the specifications in this example, but hypercontracts enable us to do so.

8.3 Representation of compsets and hypercontracts

We have laid out the theory of hypercontracts, built in three stages. We now discuss the issue of syntactically representing these objects. Up to now, we have written compsets explicitly as sets. Doing this, however, results in a problem of *portability*. Consider again the example shown in Figure 8.1. In our running example, we found out that we could express the property of non-interference for the top-level component through the expression $\{(H, S, P, O_1, O_2, O) \mid \exists f \in (2^1 \times 2^n \to 2^o).\, O = f(H, P)\}$. What would happen if we added more internal variables to the system? Suppose, for example, that we have an additional variable O_3. In that case, the theory of components needs to define component behaviors also over the variable O_3, and the compset in question becomes $\{(H, S, P, O_1, O_2, O_3, O) \mid \exists f \in (2^1 \times 2^n \to 2^o).\, O = f(H, P)\}$. This makes it clear that compsets change when the theory of components modifies its variables. Yet, we would agree that the two compsets we wrote represent the same components.

In order to have a representation of compsets which is invariant to adding new variable names to the theory of components, we assume we have a logic Ψ whose formulas are denoted by compsets. We require Ψ to be a lattice and the denotation map

$$\mathbf{Den} : \Psi \to \mathbf{CmpSet}$$

to be a lattice map. This means that $\mathbf{Den}(\psi \wedge \psi') = \mathbf{Den}(\psi) \wedge \mathbf{Den}(\psi')$ and $\mathbf{Den}(\psi \vee \psi') = \mathbf{Den}(\psi) \vee \mathbf{Den}(\psi')$. The **Den** map also provides us with the means to represent hypercontracts, as these are given by a pair of compsets.

Example 8.3.1. *As an example, suppose we have a theory with only one component: a voltage amplifier with an output O having the same real value as its input I. The component is given by $M = \{(I, O) \in \mathbb{R}^2 \mid O = I\}$. The theory of compsets has two elements: $\emptyset$ and $\{M\}$. Suppose we have a logic Ψ with symbols i, o in which the formula $\psi := i = o$ is well defined and has a denotation $\boldsymbol{Den}(\psi) = \{C \in \mathbb{M} \mid \forall (I, O) \in C.\, I = O\} = \{M\}$.*

Now suppose we alter the component theory so that it has an additional real variable T. Now the component M becomes $M' = \{(I, O, T) \in \mathbb{R}^3 \mid O = I\}$. Observe that the description of the component M has changed; yet, we could say that M' is completely independent of T. Now suppose we have a logic Ψ' with symbols i, o, t in which the formula $\psi := i = o$ is also well-defined. We can build a denotation map $\boldsymbol{Den}' : \Psi' \to \mathbf{CmpSet}$ such that $\boldsymbol{Den}'(\psi) = \{C \in \mathbb{M} \mid \forall (I, O, T) \in C.\, I = O\} = \{M'\}$. □

We observe in this example that we were able to use the same formula ψ in order to represent a compset, even when we modified the underlying symbols on which objects were defined. In other words, representations allow us to define compsets by only using "local knowledge" about the interfaces of the components described by the compset, despite the fact that components are denoted on the set of behaviors of the entire system.

8.4 Behavioral modeling

In the behavioral approach to system modeling, we start with a set $\mathcal{B}$ whose elements we call behaviors. Components are defined as subsets of $\mathcal{B}$. They contain the behaviors they can display. A component M is a subcomponent of M' if M' contains all the behaviors of M, i.e., if $M \subseteq M'$. Component composition is given by set intersection: $M \times M' \stackrel{\text{def}}{=} M \cap M'$. If we represent the components as $M = \{b \in \mathcal{B} \mid \phi(b)\}$ and $M' = \{b \in \mathcal{B} \mid \phi'(b)\}$ for some constraints ϕ and ϕ', then composition is $M \times M' = \{b \in \mathcal{B} \mid \phi(b) \wedge \phi'(b)\}$, i.e., the behaviors that simultaneously meet the constraints of M and M'. This notion of composition is independent of the connection topology: the topology is inferred from the behaviors of the components.

We will consider two contract theories we can build with these components. The first is based on unconstrained hyperproperties; the second is based on downward-closed hyperproperties.

8.4.1 Assume-guarantee contracts

Now we express AG contracts using hypercontracts. We can instantiate trace properties as a **CmpSet** lattice. Each compset is of the form $H = 2^M$ for some component $M \subseteq \mathcal{B}$. Observe that the satisfaction of a compset by a component $M' \in 2^M$ happens if and only if $M' \leq M$. The meet of two compsets $2^M \wedge 2^{M'}$ is $2^{M \cap M'} = 2^M \cap 2^{M'}$, but the join of two elements $2^M \vee 2^{M'}$ is $2^{M \cup M'} \neq 2^M \cup 2^{M'}$. The composition of two compsets is given by $2^M \times 2^{M'} = 2^{M \cap M'}$, and the quotient is $2^M / 2^{M'} = 2^{M/M'}$.

Assume-guarantee contracts are often given as a pair of trace-properties (A, G), where A states the assumptions made on the environment, and G states what the component in question should guarantee when operating in a valid environment (i.e., one that meets the assumptions). We observe that any closed system obtained using environments that meet the assumptions is restricted to $G \cap A$; thus, we set the closed-system spec to $\mathcal{S} = 2^{A \cap G}$. Define the hypercontract $\mathcal{C} = (2^A, 2^{A \cap G})$. The environments are $\mathcal{E} = 2^A$, namely, all $E \subseteq A$, and the implementations are $\mathcal{I} = 2^{(A \cap G)/A} = 2^{G/A}$, that is, all $M \subseteq G/A$. Observe that $\mathcal{S}/\mathcal{I} = \mathcal{E}$, so $\mathcal{C}$ is saturated. Now suppose we have another hypercontract $\mathcal{C}' = (2^{A'}, 2^{A' \cap G'})$ with environments $\mathcal{E}'$ and implementations $\mathcal{I}'$. We observe that $\mathcal{E} \leq \mathcal{E}'$ if and only if $A \subseteq A'$; moreover, $\mathcal{I}' \leq \mathcal{I}$ if and only if $G'/A' \leq G/A$. This means that $\mathcal{C}' \leq \mathcal{C}$ if and only if the assume-guarantee contracts (A, G) and (A', G') satisfy $(A', G') \leq (A, G)$.

Suppose we have a second hypercontract $\mathcal{C}' = (2^{A'}, 2^{G'/A'})$. Applying (8.3), we obtain a composition formula for these hypercontracts: $\mathcal{C} \parallel \mathcal{C}' = \left(2^{A'/(G/A)} \wedge 2^{A/(G'/A')}, 2^{G/A} \wedge 2^{G'/A'}\right)$, whose environments and implementations are exactly those obtained from the composition of the assume-guarantee contracts (A, G) and (A', G') [18].

Given two assume-guarantee contracts (A_i, G_i) for $i = 1, 2$, we consider the merging of their hypercontracts. We have $(2^{A_1}, 2^{A_1 \cap G_1}) \bullet (2^{A_2}, 2^{A_2 \cap G_2}) = (2^{A_1 \cap A_2}, 2^{A_1 \cap G_1 \cap A_2 \cap G_2})$. Observe that this last hypercontract has environments $2^{A_1 \cap A_2}$. The implementations of the

hypercontract are $2^{(G_1 \cap G_2)/(A_1 \cap A_2)}$. This is the definition of merging for assume-guarantee contracts [161].

8.4.2 Interval AG contracts

Now we explore AG contracts with a must modality. We will assume that elements of **CmpSet** are property intervals. In other words, if H is a compset, we can find components $L, R \in \mathbb{M}$ such that $H = \{M \in \mathbb{M} \mid L \leq M \leq R\}$. We will refer to such compsets as *modal* or *interval* compsets, and we write them as $H = [L, R]$. The name modal is used to indicate that a component satisfying a modal compset must implement some behaviors (those contained in L) and is only allowed to implement certain behaviors (those contained in R).

Let $H = [L, R]$ and $H' = [L', R']$. The operations on compsets are given by

$$\begin{aligned}
H \parallel H' &= \{M \parallel M' \mid L \leq M \leq R \text{ and } L' \leq M' \leq R'\} = [L \cap L', R \cap R'], \\
H \wedge H' &= \{M \mid L \leq M \leq R \text{ and } L' \leq M \leq R'\} = [L \cup L', R \cap R'], \\
H \vee H' &= \{M \mid L \leq M \leq R \text{ or } L' \leq M \leq R'\} = [L \cap L', R \cup R'], \text{ and} \\
H/H' &= \vee \{[L'', R''] \mid H' \parallel [L'', R''] \leq H\} \\
&= \vee \{[L'', R''] \mid [L' \cap L'', R' \cap R''] \leq H\} \\
&= [L, R \cup \neg R'] \quad \text{(only defined when } L \leq L'\text{)}.
\end{aligned}$$

We now state the expressions for composition and quotient.

Proposition 8.4.1. *Suppose $\mathcal{C} = (\mathcal{E}, \mathcal{I})$ and $\mathcal{C}' = (\mathcal{E}', \mathcal{I}')$ with $\mathcal{E} = [L_e, R_e]$, $\mathcal{I} = [L_i, R_i]$, $\mathcal{E}' = [L'_e, R'_e]$, and $\mathcal{I}' = [L'_i, R'_i]$. The composition of these hypercontracts is only defined when $L_e = L_i = L'_e = L'_i$. Set $L = L_e$. Then the composition $\mathcal{C} \parallel \mathcal{C}' = (\mathcal{E}_c, \mathcal{I}_c)$ is of the form*

$$\begin{aligned}
\mathcal{E}_c &= [L, (R_e \cap R'_e) \cup (R'_e \cap \neg R'_i) \cup (R_e \cap \neg R_i)] \text{ and} \\
\mathcal{I}_c &= [L, (R_i \cup \neg R_e) \cap (R'_i \cup \neg R'_e)].
\end{aligned}$$

Now suppose $\mathcal{C} = (\mathcal{E}, \mathcal{I})$ and $\mathcal{C}'' = (\mathcal{E}'', \mathcal{I}'')$ with $\mathcal{E}'' = [L''_e, R''_e]$, $\mathcal{I}'' = [L''_i, R''_i]$, $\mathcal{E} = [L_e, R_e]$, and $\mathcal{I} = [L_i, R_i]$. The residual $\mathcal{C}''/\mathcal{C} = (\mathcal{E}_r, \mathcal{I}_r)$ is only defined when $L''_i \leq L_i = L_e \leq L''_e$. Call $L = L_i$. The components of the quotient have the form

$$\begin{aligned}
\mathcal{E}_r &= [L, R''_e \cap (R_i \cup \neg R_e)] \text{ and} \\
\mathcal{I}_r &= [L, (R_e \cap R''_i) \cup \neg R''_e \cup (R_e \cap \neg R_i)].
\end{aligned}$$

Proof. We consider contract composition. Let $\mathcal{C} = (\mathcal{E}, \mathcal{I})$ and $\mathcal{C}' = (\mathcal{E}', \mathcal{I}')$ with $\mathcal{E} = [L_e, R_e]$, $\mathcal{I} = [L_i, R_i]$, $\mathcal{E}' = [L'_e, R'_e]$, and $\mathcal{I}' = [L'_i, R'_i]$. Their composition of these two contracts $\mathcal{C} \parallel \mathcal{C}' = (\mathcal{E}_c, \mathcal{I}_c)$ requires us to compute

$$\mathcal{E}_c = (\mathcal{E}'/(\mathcal{I}/\mathcal{E})) \wedge (\mathcal{E}/(\mathcal{I}'/\mathcal{E}')) \text{ and } \mathcal{I}_c = ((\mathcal{I}/\mathcal{E}) \parallel (\mathcal{I}'/\mathcal{E}'))/\mathcal{E}_c.$$

Since we have to compute $\mathcal{I}/\mathcal{E}$, we must have $L_i \leq L_e$. Similarly, to compute $\mathcal{I}'/\mathcal{E}'$, we need $L'_i \leq L'_e$. Now, to compute $\mathcal{E}'/(\mathcal{I}/\mathcal{E})$ and $\mathcal{E}/(\mathcal{I}'/\mathcal{E}')$, we must have $L'_e \leq L_i$ and $L_e \leq L'_i$. Then

$$L'_e \leq L_i \leq L_e \text{ and } L_e \leq L'_i \leq L'_e,$$

so we must have $L_e = L_i = L'_i = L'_e$ for contract composition to be well defined. To simplify notation, let $L = L_e = L_i = L'_i = L'_e$. We obtain

$$\begin{aligned}\mathcal{E}_c &= (\mathcal{E}'/(\mathcal{I}/\mathcal{E})) \wedge (\mathcal{E}/(\mathcal{I}'/\mathcal{E}')) = (\mathcal{E}'/([L, R_i \cup \neg R_e])) \wedge (\mathcal{E}/([L, R'_i \cup \neg R'_e])) \\ &= [L, R'_e \cup \neg (R_i \cup \neg R_e)] \wedge [L, R_e \cup \neg (R'_i \cup \neg R'_e)] \\ &= [L, (R_e \cap R'_e) \cup (R'_e \cap \neg R'_i) \cup (R_e \cap \neg R_i)]\end{aligned}$$

and

$$\begin{aligned}\mathcal{I}_c &= ((\mathcal{I}/\mathcal{E}) \parallel (\mathcal{I}'/\mathcal{E}'))/\mathcal{E}_c = ([L, R_i \cup \neg R_e] \parallel [L, R'_i \cup \neg R'_e])/\mathcal{E}_c \\ &= ([L, (R_i \cup \neg R_e) \cap (R'_i \cup \neg R'_e)])/\mathcal{E}_c = [L, (R_i \cup \neg R_e) \cap (R'_i \cup \neg R'_e)].\end{aligned}$$

Finally, we seek expressions for the residual. Let $\mathcal{C} = (\mathcal{E}, \mathcal{I})$ and $\mathcal{C}'' = (\mathcal{E}'', \mathcal{I}'')$ with $\mathcal{E}'' = [L''_e, R''_e]$, $\mathcal{I}'' = [L''_i, R''_i]$, $\mathcal{E} = [L_e, R_e]$, and $\mathcal{I} = [L_i, R_i]$. The residual $\mathcal{C}''/\mathcal{C} = (\mathcal{E}_r, \mathcal{I}_r)$ is given by

$$\mathcal{E}_r = \mathcal{E}'' \parallel (\mathcal{I}/\mathcal{E}) \text{ and } \mathcal{I}_r = ((\mathcal{E}/\mathcal{E}'') \wedge ((\mathcal{I}''/\mathcal{E}'')/(\mathcal{I}/\mathcal{E})))/\mathcal{E}_r.$$

To compute $\mathcal{I}/\mathcal{E}$, $\mathcal{E}/\mathcal{E}''$, and $\mathcal{I}''/\mathcal{E}''$, we must have

$$L_i \leq L_e \leq L''_e, \text{ and } L''_i \leq L''_e.$$

We compute

$$\begin{aligned}&\mathcal{E}_r = \mathcal{E}'' \parallel [L_i, R_i \cup \neg R_e] = [L_i, R''_e \cap (R_i \cup \neg R_e)] \text{ and} \\ &\mathcal{I}_r = ((\mathcal{E}/\mathcal{E}'') \wedge ((\mathcal{I}''/\mathcal{E}'')/(\mathcal{I}/\mathcal{E})))/\mathcal{E}_r = \\ &([L_e, R_e \cup \neg R''_e] \wedge ([L''_i, R''_i \cup \neg R''_e]/[L_i, R_i \cup \neg R_e]))/\mathcal{E}_r.\end{aligned}$$

We have the further constraint $L''_i \leq L_i$. Thus, $L''_i \leq L_i \leq L_e \leq L''_e$ and

$$\begin{aligned}\mathcal{I}_r =& ([L_e, R_e \cup \neg R''_e] \wedge [L''_i, (R''_i \cup \neg R''_e) \cup (R_e \cap \neg R_i)])/\mathcal{E}_r \\ =& [L_e \cup L''_i, (R_e \cup \neg R''_e) \cap ((R''_i \cup \neg R''_e) \cup (R_e \cap \neg R_i))]/\mathcal{E}_r \\ =& [L_e, (R_e \cap R''_i) \cup \neg R''_e \cup (R_e \cap \neg R_i)]/[L_i, R''_e \cap (R_i \cup \neg R_e)].\end{aligned}$$

We have the additional constrraint $L_e \leq L_i$. Thus, we have $L_e = L_i = L$, and we have $L''_i \leq L \leq L''_e$ and

$$\mathcal{I}_r = [L, (R_e \cap R''_i) \cup \neg R''_e \cup (R_e \cap \neg R_i)].$$

□

8.4.3 General hypercontracts

The most expressive behavioral theory of hypercontracts is obtained when we place no restrictions on the structure of compsets and hypercontracts. In this case, the elements of **CmpSet** are all objects $H \in 2^{2^{\mathcal{B}}}$, i.e., all hyperproperties. The meet and join of compsets are set intersection and union, respectively, and their composition and quotient are given by (8.1) and (8.2), respectively. Hypercontracts are of the form $\mathcal{C} = (\mathcal{E}, \mathcal{I})$ with all extrema achieved in the binary operations, i.e., for a second hypercontract $\mathcal{C}' = (\mathcal{E}', \mathcal{I}')$, the meet, join, and composition (8.3) are, respectively, $\mathcal{C} \wedge \mathcal{C}' = (\mathcal{E} \cup \mathcal{E}', \mathcal{I} \cap \mathcal{I}')$, $\mathcal{C} \vee \mathcal{C}' = (\mathcal{E} \cap \mathcal{E}', \mathcal{I} \cup \mathcal{I}')$, and $\mathcal{C} \parallel \mathcal{C}' = \left(\frac{\mathcal{E}'}{\mathcal{I}} \cap \frac{\mathcal{E}}{\mathcal{I}'}, \mathcal{I} \parallel \mathcal{I}'\right)$. From these follow the operations of quotient, and merging.

8.4.4 Conic (or downward-closed) hypercontracts

We assume that **CmpSet** contains exclusively downward-closed hyperproperties. Let $H \in$ **CmpSet**. We say that $M \models H$ is a maximal component of H when H contains no set bigger than M, i.e., if $\forall M' \models H.\ M \leq M' \Rightarrow M' = M$.

We let $\overline{H}$ be the set of maximal components of H:

$$\overline{H} = \{M \models H \mid \forall M' \models H.\ M \leq M' \Rightarrow M' = M\}.$$

Due to the fact H is downward-closed, the set of maximal components is a unique representation of H. We can express H as

$$H = \bigcup_{M \in \overline{H}} 2^M.$$

We say that H is *k-conic* if the cardinality of $\overline{H}$ is finite and equal to k, and we write this

$$H = \langle M_1, \ldots, M_k \rangle, \text{ where } \overline{H} = \{M_1, \ldots, M_k\}.$$

8.4.4.1 Order

The notion of order on **CmpSet** can be expressed using this notation as follows: suppose $H' = \langle M' \rangle_{M' \in \overline{H}'}$. Then

$$H' \leq H \text{ if and only if } \forall M' \in \overline{H}'\ \exists M \in \overline{H}.\ M' \leq M.$$

8.4.4.2 Composition

Composition in **CmpSet** becomes

$$H \times H' = \bigcup_{\substack{M \in \overline{H} \\ M' \in \overline{H}'}} 2^{M \cap M'} = \langle M \cap M' \rangle_{\substack{M \in \overline{H} \\ M' \in \overline{H}'}}. \tag{8.5}$$

Therefore, if H and H' are, respectively, k- and k'-conic, $H \times H'$ is at most kk'-conic.

8.4.4.3 Quotient

Suppose H_q satisfies

$$H' \times H_q \leq H.$$

Let $M_q \in \overline{H}_q$. We must have

$$M_q \times M' \models H \text{ for every } M' \in \overline{H}',$$

which means that for each $M' \in \overline{H}'$ there must exist an $M \in \overline{H}$ such that $M_q \times M' \leq M$; let us denote by $M(M')$ a choice $M' \mapsto M$ satisfying this condition. Therefore, we have

$$M_q \leq \bigwedge_{M' \in \overline{H}'} \tfrac{M(M')}{M'}, \tag{8.6}$$

Clearly, the largest such M_q is obtained by making (8.6) an equality. Thus, the cardinality of the quotient is bounded from above by $k^{k'}$ since we have

$$H_q = \left\langle \bigwedge_{M' \in \overline{H}'} \tfrac{M(M')}{M'} \right\rangle_{\substack{M(M') \in \overline{H} \\ \forall M' \in \overline{H}'}}. \tag{8.7}$$

8.4.4.4 Contracts

Now we assume that the objects of **CmpSet** are pairs of *downward-closed compsets.* If we have two hypercontracts $\mathcal{C} = (\mathcal{E}, \mathcal{I})$ and $\mathcal{C}' = (\mathcal{E}', \mathcal{I}')$, their composition and quotient are, respectively,

$$\mathcal{C} \parallel \mathcal{C}' = \left(\frac{\mathcal{E}}{\mathcal{I}'} \wedge \frac{\mathcal{E}'}{\mathcal{I}}, \mathcal{I} \times \mathcal{I}' \right) \text{ and } \mathcal{C}/\mathcal{C}' = \left(\mathcal{E} \times \mathcal{I}', \frac{\mathcal{I}}{\mathcal{I}'} \wedge \frac{\mathcal{E}'}{\mathcal{E}} \right). \tag{8.8}$$

8.5 Receptive languages and interface hypercontracts

In this section we connect the notion of a hypercontract with specifications expressed as interface automata [54]. With interface theories, we bring in the notion of input-output profiles as an extra typing for components—so far, this was not considered in our development. This effectively partitions $\mathbb{M}$ into sets containing components sharing the same profile.

Our theory of components is constructed from a new notion called *receptive languages.* These objects can be understood as the trace denotations of receptive I/O automata [136]. We will consider downward-closed, 1-conic compsets, see Section 8.4.4. And interface hypercontracts will be pairs of these with a very specific structure. At the end of the section we show how the denotation of interface automata is captured by interface hypercontracts. One novelty of our approach is that the computation of the composition of hypercontracts, which matches that of interface automata (as we will see), is inherited from our general theory by specializing the component and compset operations.

8.5.1 The components are receptive languages

Fix once and for all an alphabet Σ. When we operate on words of Σ^*, we will use $\circ$ for word concatenation, and we'll let $\mathrm{Pre}(w)$ be the set of prefixes of a word w. These operations are extended to languages:

$$L \circ L' = \{w \circ w' \mid w \in L \text{ and } w' \in L'\},$$

and $\mathrm{Pre}(L) = \bigcup_{w \in L} \mathrm{Pre}(w)$. An *input-output* signature of Σ (or simply an IO signature when the alphabet is understood), denoted (I, O), is a partition of Σ in sets I and O, i.e., I and O are disjoint sets whose union is Σ.

Definition 8.5.1. *Let (I, O) be an IO signature. A language L of Σ is an I-receptive language if*

- *L is prefix-closed; and*
- *if $w \in L$ and $w' \in I^*$ then $w \circ w' \in L$.*

The set of all I-receptive languages is denoted $\mathcal{L}_I$.

Proposition 8.5.2. *Let (I, O) be an IO signature. Then $\mathcal{L}_I$ is closed under intersection and union.*

Under the subset order, $\mathcal{L}_I$ is a lattice with intersection as the meet and union as the join. Further, the smallest and largest elements of $\mathcal{L}_I$ are, respectively, $0 = I^*$ and $1 = \Sigma^*$. It so happens that $\mathcal{L}_I$ is a Heyting algebra. To prove this, it remains to be shown that it has exponentiation (i.e., that the meet has a right adjoint).

Proposition 8.5.3. *Let $L, L' \in \mathcal{L}_I$. The object*

$$L' \to L = \{w \in \Sigma^* \mid \mathit{Pre}(w) \cap L' \subseteq L\}$$

is an element of $\mathcal{L}_I$ and satisfies the property of the exponential.

We further explore the structure of the exponential. To do this, it will be useful to define the following set: for languages L, L' and a set $\Gamma \subseteq \Sigma$, we define the set of *missing Γ-extensions of L' with respect to L* as

$$\mathrm{MissExt}(L, L', \Gamma) = (((L \cap L') \circ \Gamma) - L') \circ \Sigma^*.$$

The elements of this set are all words of the form $w \circ \sigma \circ w'$, where $w \in L \cap L'$, $\sigma \in \Gamma$, and $w' \in \Sigma^*$. These words satisfy the condition $w \circ \sigma \notin L'$. In other words, we find the words of $L \cap L'$ which, when extended by a symbol of Γ, leave the language L', and extend these words by the symbols that make them leave L' and then by every possible word of Σ^*.

Proposition 8.5.4. *Let $L, L' \in \mathcal{L}_I$. The exponential is given by*

$$L' \to L = L \cup \mathit{MissExt}(L, L', O).$$

At this point, it has been established that each $\mathcal{L}_I$ is a Heyting algebra. Now we move to composition and quotient, which involve languages of different IO signatures.

8.5.2 Composition and quotient of receptive languages

To every $I \subseteq \Sigma$, we have associated the set of languages $\mathcal{L}_I$. Suppose $I' \subseteq I$. Then $L \in \mathcal{L}_I$ if it is prefix-closed, and the extension of any word of L by any word of I^* remains in L. But since $I' \subseteq I$, this means that the extension of any word of L by any word of $(I')^*$ remains in L, so $L \in \mathcal{L}_{I'}$. We have shown that $I \subseteq I' \Rightarrow \mathcal{L}_{I'} \leq \mathcal{L}_I$. Thus, the map $I \mapsto \mathcal{L}_I$ is a contravariant functor $2^\Sigma \to 2^{2^{\Sigma^*}}$.

Since $I' \subseteq I$ implies that $\mathcal{L}_I \leq \mathcal{L}_{I'}$, we define the embedding $\iota : \mathcal{L}_I \to \mathcal{L}_{I'}$ which maps a language of $\mathcal{L}_I$ to the same language, but interpreted as an element of $\mathcal{L}_{I'}$,

Let (I, O) and (I', O') be IO signatures of Σ, $L \in \mathcal{L}_I$, and $L' \in \mathcal{L}_{I'}$. The composition of structures with labeled inputs and outputs traditionally requires that objects to be composed can't share outputs. We say that IO signatures (I, O) and (I', O') are *compatible* when $O \cap O' = \emptyset$. This is equivalent to requiring that $I \cup I' = \Sigma$. Moreover, the object generated by the composition should have as outputs the union of the outputs of the objects being composed. This reasoning leads us to the definition of composition:

Definition 8.5.5 (composition)**.** *Let (I, O) and (I', O') be compatible IO signatures of Σ. Let $L \in \mathcal{L}_I$ and $L' \in \mathcal{L}_{I'}$. The operation of language* composition*, $\times : \mathcal{L}_I, \mathcal{L}_{I'} \to \mathcal{L}_{I \cap I'}$, is given by*

$$L \times L' = \iota L \wedge \iota' L',$$

for the embeddings $\iota : \mathcal{L}_I \to \mathcal{L}_{I \cap I'}$ and $\iota' : \mathcal{L}_{I'} \to \mathcal{L}_{I \cap I'}$.

The adjoint of this operation is the quotient. We will investigate when the quotient is defined. Let $I, I' \subseteq \Sigma$ with $I \subseteq I'$, $L \in \mathcal{L}_I$, and $L' \in \mathcal{L}_{I'}$. Suppose there is $I_r \subseteq \Sigma$ such that the composition rule $\times : \mathcal{L}_{I'}, \mathcal{L}_{I_r} \to \mathcal{L}_I$ is defined. This means that $I' \cup I_r = \Sigma$ and $I' \cap I_r = I$. Solving yields $I_r = I \cup \neg I' = I \cup O'$.

Observe that the smallest element of $\mathcal{L}_{I_r}$ is I_r^*. Thus, the existence of a language $L'' \in \mathcal{L}_{I_r}$ such that $L'' \times L' \leq L$ requires that $L' \cap I_r^* \subseteq L$. Clearly, not every pair L, L' satisfies this property since we can take, for example, $L = I^*$ and $L' = \Sigma^*$ to obtain $L' \cap I_r^* = (I \cup O')^* \not\subseteq I^*$, provided $I' \neq \Sigma$.

We proceed to obtain a closed-form expression for the quotient, but first we define a new operator. For languages L, L' and sets $\Gamma, \Delta \subseteq \Sigma$, the following set of (L', Γ, Δ)-*uncontrollable extensions of $L \cap L'$*

$$\mathrm{Unc}(L, L', \Gamma, \Delta) = \left\{ w \in L \cap L' \;\middle|\; \begin{array}{l} \exists w' \in (\Gamma \cup \Delta)^* \;\wedge\; \sigma \in \Gamma. \\ w \circ w' \in L \cap L' \wedge \\ w \circ w' \circ \sigma \in L' - L \end{array} \right\} \circ \Sigma^*. \tag{8.9}$$

contains: (i) all words of $L \cap L'$ which can be uncontrollably extended to a word of $L' - L$ by appending a word of $(\Gamma \cup \Delta)^*$ and a symbol of Γ, and (ii) all suffixes of such words. Equivalently, $\mathrm{Unc}(L, L', \Gamma, \Delta)$ contains all extensions of the words $w \in L \cap L'$ such that there

are extensions of w by words $w' \in (\Gamma \cup \Delta)^*$ that land in L' but not in L after appending to the extensions $w \circ w'$ a symbol of Γ.

Proposition 8.5.6. *Let (I, O) and (I', O') be IO signatures of Σ such that $I \subseteq I'$. Let $L \in \mathcal{L}_I$ and $L' \in \mathcal{L}_{I'}$. Let $I_r = I \cup O'$, and assume that $L' \cap I_r^* \subseteq L$. Then the largest $L'' \in \mathcal{L}_{I_r}$ such that $L'' \parallel L' \leq L$ is denoted L/L' and is given by*

$$L/L' = (L \cap L' \cup \mathit{MissExt}(L, L', O')) - \mathit{Unc}(L, L', O', I).$$

We have defined receptive languages together with a preorder and a composition operation with its adjoint. These objects will constitute our theory of components, i.e., $\mathbb{M} = \oplus_{I \in 2^\Sigma} \mathcal{L}_I$.

8.5.3 Compsets and interface hypercontracts

Using the set of components just defined, we proceed to build compsets and hypercontracts. The compsets contain components adhering to the same IO signature. Thus, again the notion of an IO signature will partition the set of compsets (and the same will happen with hypercontracts). This means that for every compset H, there will always be an $I \subseteq \Sigma$ such that $H \subseteq \mathcal{L}_I$.

For $I \subseteq \Sigma$, and $L \in \mathcal{L}_I$, we will consider compsets of the form

$$\left\{ M \in 2^L \;\middle|\; I^* \subseteq M \right\}, \text{ denoted by } [0, L],$$

where 0 is I^*, the smallest element of $\mathcal{L}_I$, i.e., the compsets are all I-receptive languages smaller than L. We will focus on hypercontracts whose implementations have signature (I, O) and whose environments have (O, I). Thus, hypercontracts will consist of pairs $\mathcal{C} = (\mathcal{E}, \mathcal{S})$ of O- and $\emptyset$-receptive compsets, respectively. We will let

$$\mathcal{S} = [0, S] = \{M \in \mathcal{L}_\emptyset \mid M \subseteq S\}$$

for some $S \in \mathcal{L}_\emptyset$. We will restrict the environments $E \in \mathcal{E}$ to those that never extend a word of S by an input symbol that S does not accept. The largest such environment is given by

$$E_S = S \cup \mathrm{MissExt}(S, S, O). \tag{8.10}$$

Since S is prefix-closed, so is E_S. Moreover, observe that E_S adds to S all those strings that are obtained by continuations of words of S by an output symbol that S does not produce. This makes E_S O-receptive. The set of environments is thus $\mathcal{E} = [O^*, E_S]$.

Having obtained the largest environment, we can find the implementations. These are given by $\mathcal{I} = [I^*, M_S]$ for $M_S = S/E_S$. Plugging the definition, we have

$$M_S = (S \cap E_S \cup \mathrm{MissExt}(S, E_S, I)) - \mathrm{Unc}(S, E_S, I, \emptyset).$$

There is no word of I^* which can extend a word of S into $E_S - S$. Thus,

$$M_S = S \cup \text{MissExt}(S, S, I).$$

Observe that S and $\text{MissExt}(S, S, I)$ are disjoint (same for $\text{MissExt}(S, S, O)$). Thus, $E_S \times M_S = S$. In summary, we observe that our hypercontracts are highly structured. They are in 1-1 correspondence with a language $S \in \mathcal{L}_\emptyset$ and an input alphabet $I \subseteq \Sigma$, i.e., there is a set isomorphism

$$\mathcal{L}_\emptyset, 2^\Sigma \xrightarrow{\sim} \mathbf{Contr}\,. \tag{8.11}$$

Indeed, given S and I, we build E_S by extending S by $\Sigma - I = O$, and M_S by extending S by I. After this, the hypercontract has environments, closed systems, and implementations $[O^*, E_S]$, $[\emptyset, S]$, and $[I^*, M_S]$, respectively.

8.5.4 Hypercontract composition

Let $S, S' \in \mathcal{L}_\emptyset$. We consider the composition of the interface hypercontracts $\mathcal{C}_R = \mathcal{C}_S \parallel \mathcal{C}_{S'}$, where $\mathcal{C}_S = ([0, E_S], [0, S])$, $\mathcal{C}_{S'} = ([0, E_{S'}], [0, S'])$ and E_S and $E_{S'}$ have signatures (O, I) and (O', I'), respectively. From the structure of interface hypercontracts, we have the relations

$$\begin{aligned} E_S &= S \cup \text{MissExt}(S, S, O) \text{ and} \\ E_{S'} &= S' \cup \text{MissExt}(S', S', O'). \end{aligned}$$

Moreover, the implementations of C, C' are, respectively, $\mathcal{I} = [I^*, M_S]$ and $\mathcal{I}' = [I'^*, M_{S'}]$, where

$$\begin{aligned} M_S &= S \cup \text{MissExt}(S, S, I) \text{ and} \\ M_{S'} &= S' \cup \text{MissExt}(S', S', I'). \end{aligned}$$

The composition of these hypercontracts is defined if (I, O) and (I', O') have compatible signatures. Suppose $\mathcal{C}_R = \mathcal{C}_S \parallel \mathcal{C}_{S'} = ([0, E_R], [0, R])$ for some $R \in \mathcal{L}_\emptyset$. Then the environments must have signature $(O \cup O', I \cap I')$, and the implementations $(I \cap I', O \cup O')$.

Finally, as usual, $E_R = R \cup \text{MissExt}(R, R, O \cup O')$ and $M_R = R \cup \text{MissExt}(R, R, I \cap I')$ are the maximal environment and implementation. R is determined as follows:

Proposition 8.5.7. *Let $\mathcal{C}_S$ and $\mathcal{C}_{S'}$ be interface hypercontracts and let $\mathcal{C}_R \stackrel{def}{=} \mathcal{C}_S \parallel \mathcal{C}_{S'}$. Then R is given by the expression*

$$R = (S \cap S') - [\mathit{Unc}(S', S, O, O') \cup \mathit{Unc}(S, S', O', O)]\,.$$

The quotient for interface hypercontracts follows from Proposition 8.2.6.

8.5.5 Connection with interface automata

Now we explore the relation of interface hypercontracts with interface automata. Let (I, O) be an IO signature. An I-interface automaton [54] is a tuple $A = (Q, q_0, \to)$, where Q is a finite set whose elements we call states, $q_0 \in Q$ is the initial state, and $\to \subseteq Q \times \Sigma \times Q$ is a deterministic transition relation (there is at most one next state for every symbol of Σ). We let $\mathcal{A}_I$ be the class of I-interface automata, and $\mathcal{A} = \oplus_{I \in 2^\Sigma} \mathcal{A}_I$. In the language of interface automata, input and output symbols are referred to as *actions*.

Given two interface automata (IA) $A_i = (Q_i, q_{i,0}, \to_i) \in \mathcal{A}_I$ for $i \in \{1, 2\}$, we say that the state $q_1 \in Q_1$ refines $q_2 \in Q_2$, written $q_1 \leq q_2$, if

- $\forall \sigma \in O, q_1' \in Q_1.\ q_1 \xrightarrow{\sigma}_1 q_1' \Rightarrow \exists q_2' \in Q_2.\ q_2 \xrightarrow{\sigma}_2 q_2'$ and $q_1' \leq q_2'$ and
- $\forall \sigma \in I, q_2' \in Q_2.\ q_2 \xrightarrow{\sigma}_2 q_2' \Rightarrow \exists q_1' \in Q_1.\ q_1 \xrightarrow{\sigma}_1 q_1'$ and $q_1' \leq q_2'$.

We say that A_1 refines A_2, written $A_1 \leq A_2$, if $q_{1,0} \leq q_{2,0}$. This defines a preorder in $\mathcal{A}_I$.

8.5.5.1 Mapping to interface hypercontracts

Suppose $A = (Q, q_0, \to) \in \mathcal{A}_I$. We define the language of A, denoted $\ell(A)$, as the set of words obtained by "playing out" the transition relation, i.e.,

$$\ell(A) = \left\{ \sigma_0 \sigma_1 \dots \sigma_n \;\middle|\; \exists q_1, \dots, q_{n-1}.\ q_i \xrightarrow{\sigma_i} q_{i+1} \text{ for } 0 \leq i < n \right\}.$$

Since $\ell(A)$ is prefix-closed, it is an element of $\mathcal{L}_\emptyset$.

From Section 8.5.3, we know that interface hypercontracts are isomorphic to a language S of $\mathcal{L}_\emptyset$ and an IO signature I. The operation $A \mapsto \ell(A)$ maps an I-receptive interface automaton A to a language of $\mathcal{L}_\emptyset$. Composing this map with the map (8.11) discussed in Section 8.5.3, we have maps $\mathcal{A} \to \mathcal{L}_\emptyset, 2^\Sigma \xrightarrow{\sim} \mathbf{Contr}$.

Thus, the interface hypercontract associated to $A \in \mathcal{A}_I$ is $\mathcal{C}_A = ([0, E_{\ell(A)}], [0, \ell(A)])$, where $E_{\ell(A)} \in \mathcal{L}_O$ is given by (8.10). The following result tells us that refinement of interface automata is equivalent to refinement of their associated hypercontracts.

Proposition 8.5.8. *Let $A_1, A_2 \in \mathcal{A}_I$. Then $A_1 \leq A_2$ if and only if $\mathcal{C}_{A_1} \leq \mathcal{C}_{A_2}$.*

8.5.5.2 Composition

Let $A_1 = (Q_1, q_{1,0}, \to_1) \in \mathcal{A}_{I_1}$ and $A_2 = (Q_2, q_{2,0}, \to_2) \in \mathcal{A}_{I_2}$. The composition of the two IA is defined if $I_1 \cup I_2 = \Sigma$. In that case, the resulting IA, $A_1 \parallel A_2$, has IO signature $(I_1 \cap I_2, O_1 \cup O_2)$. The elements of the composite IA are $(Q, (q_{1,0}, q_{2,0}), \to_c)$, where the set of states and the transition relation are obtained through the following algorithm:

- Initialize $Q \coloneqq Q_1 \times Q_2$. For every $\sigma \in \Sigma$, $(q_1, q_2) \xrightarrow{\sigma}_c (q_1', q_2')$ if $q_1 \xrightarrow{\sigma}_1 q_1'$ and $q_2 \xrightarrow{\sigma}_2 q_2'$.

• Initialize the set of invalid states to those states where one interface automaton can generate an output action which the other interface automaton does not accept:

$$N \coloneqq \left\{ (q_1, q_2) \in Q_1 \times Q_2 \;\middle|\; \begin{array}{l} \exists\, q_2' \in Q_2, \sigma \in O_2\, \forall q_1' \in Q_1. \\ \quad q_2 \xrightarrow{\sigma}_2 q_2' \wedge \neg \left(q_1 \xrightarrow{\sigma}_1 q_1' \right) \text{ or} \\ \exists\, q_1' \in Q_1, \sigma \in O_1\, \forall q_2' \in Q_2. \\ \quad q_1 \xrightarrow{\sigma}_1 q_1' \wedge \neg \left(q_2 \xrightarrow{\sigma}_2 q_2' \right) \end{array} \right\}.$$

• Also deem invalid a state such that an output action of one of the interface automata makes a transition to an invalid state, i.e., iterate the following rule until convergence:

$$N \coloneqq N \cup \left\{ (q_1, q_2) \in Q_1 \times Q_2 \;\middle|\; \begin{array}{l} \exists\, (q_1', q_2') \in N, \sigma \in O_1 \cup O_2. \\ (q_1, q_2) \xrightarrow{\sigma}_c (q_1', q_2') \end{array} \right\}.$$

• Now remove the invalid states from the IA:

$$\begin{aligned} Q &\coloneqq Q - N \text{ and} \\ \rightarrow_c &\coloneqq \rightarrow_c - \{ (q, \sigma, q') \in \rightarrow_c \mid q \in N \text{ or } q' \in N \}. \end{aligned}$$

It turns out that composing IA is equivalent to composing their associated hypercontracts:

Proposition 8.5.9. *Let $A_1, A_2 \in \mathcal{A}_I$. Then $\mathcal{C}_{A_1 \| A_2} = \mathcal{C}_{A_1} \parallel \mathcal{C}_{A_2}$.*

Propositions 8.5.8 and 8.5.9 express that our model of interface hypercontracts is equivalent to Interface Automata. We observe that the definition for the parallel composition of interface hypercontracts is straightforward, unlike for the Interface Automata (the latter involves the iterative pruning of invalid states). In fact, in our case this pruning is hidden behind the formula (8.9) defining the set Unc().

8.6 Summary

We proposed hypercontracts, a generic model of contracts providing a richer algebra than the metatheory of [19]. Hypercontracts keep the separation of assumptions and guarantees explicit, as we have found this is the way that is most intuitive for applications. We started from a generic model of components equipped with a simulation preorder and parallel composition. On top of them, we considered compsets (or hyperproperties, for behavioral formalisms), which are lattices of sets of components equipped with parallel composition and quotient; compsets are our generic model formalizing "properties." Hypercontracts are then defined as pairs of compsets specifying the allowed environments and either the obligations of the closed system or the set of allowed implementations—both forms are useful.

We specialized hypercontracts by restricting them to conic hypercontracts, whose environments and closed systems are described by a finite number of components. Conic hypercontracts include assume-guarantee contracts as a specialization. Specializing them in a different direction provided us with a compact and elegant model of interface hypercontracts, which are conic hypercontracts built on top of input/output components specified as receptive languages. We showed that interface hypercontracts coincide with interface automata; however, our formulas for the parallel composition are direct and do not need the iterative procedure of state pruning, needed in interface automata. We illustrated the versatility of our model on the definition of contracts for information flow in security and robustness of data-driven components.

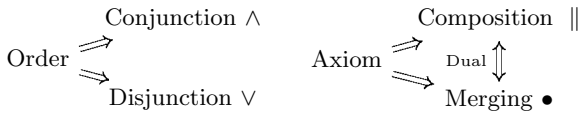
Chapter 9

Conclusions

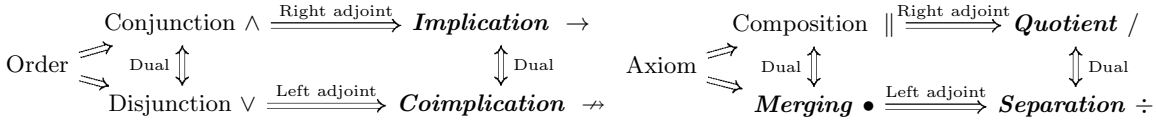
This thesis began by considering contracts defined as pairs of properties coming from a universe of behaviors. We studied the binary operations that exist on AG contracts. We then studied algebraic structures contained in an AG algebra arising from any Boolean algebra. After considering syntactic issues in contract manipulations, we extended AG reasoning to hyperproperties by introducing hypercontracts.

Algebraic aspects of AG contracts

The immediate predecessors of our work on assume-guarantee contracts from an algebraic standpoint are [18, 19, 154]. The following diagram shows the known contract operations from these papers, and the way these operations are related. Conjunction and disjunction come from a notion of order, and composition is proposed by axiom.



We list merging as a known operation because its closed form appears in [154]. One of our contributions was to realize that this operation has a role to play in merging viewpoints [161]. After the work reported in this thesis, the diagram is as follows:



We identified four contract monoids in a contract algebra and realized that they are all isomorphic. We identified four semirings in the contract algebra and realized that there are two isomorphic pairs. It remains to be verified whether an additional isomorphism exists. The more isomorphisms there are, the less work is needed to characterize maps between

algebraic contract structures arising from different Boolean algebras. The characterization of maps between contract monoids is complete. Work remains to be done on the characterization of semiring and partial order morphisms between contract structures. The last point is particularly relevant for a general characterization of Galois connections for contracts.

Our identification of contract actions opened the door to linear algebra on AG contracts. The application of a contract linear algebra in a design methodology is material for future work.

Algebraic structures that resemble assume-guarantee contracts have recently been studied by universal algebraists. In some of their work, quotients are provided for certain notions of composition [70]. The link between these theories and contracts has not been explored yet.

Methodology and tools

Chapter 7 provided some considerations for the implementation of the algebraic operations of contracts. A key observation was that the contract operations are all defined as an optimum. This does not mean that they are the final product that we want to provide to an engineer or to a tool for the next processing step, as shown in our examples. The implementation of tools based on these ideas is a matter of future work.

Our description of algebraic operations did not consider the issue of the original generation of the contracts. We assumed we had contracts available for us to compose, merge, etc. Inductive synthesis has enjoyed success in the inference of specifications, and we anticipate it will continue to play a key role in contract-based design [92, 99, 182].

Results on contracts have also been recently obtained in the domain of control systems. Kim et al. [106] connect assume-guarantee reasoning with small-gain theorems for compositional design. Phan-Minh and Murray [164, 165] introduce the notion of reactive contracts. Saoud et al. [178, 179] propose a framework of assume-guarantee contracts for input/output discrete or continuous time systems. Assumptions vs. guarantees are properties stated on inputs vs. outputs; with this restriction, reactive contracts are considered and an elegant formula is proposed for the parallel composition of contracts. Shali et al. [184] use linear systems to express the assumptions and guarantees of dynamical systems. They define a conjunction operation to merge specifications and a refinement relation to compare them. The unification of the algebra of contracts presented in this thesis with these lines of work remains to be effected.

Hypercontracts

Hypercontracts extend the scope of AG reasoning to structured hyperproperties. The flexibility and power of hypercontracts suggests that a number of directions that were opened in the standard contract reference [19], but not explored to their end, can now be re-investigated with more powerful tools: contracts and testing, subcontract synthesis (for requirement en-

gineering), contracts and abstract interpretation, and contracts in physical system modeling. Indeed, the Simulink and Modelica tools propose requirements toolboxes in which requirements are physical system properties that can be tested on a given system model, thus providing a limited form of contract. This motivates the development of a richer contract framework helping for requirement engineering in Cyber-Physical Systems design.

Bibliography

[1] Abadi, M., and Lamport, L. Composing specifications. *ACM Transactions on Programming Languages and Systems 15*, 1 (Jan. 1993), 73–132.

[2] Adam, E. M. *Systems, generativity and interactional effects.* PhD thesis, Massachusetts Institute of Technology, 2017.

[3] Adam, E. M., and Dahleh, M. A. On the abstract structure of the behavioral approach to systems theory. *arXiv preprint arXiv:1911.10398* (2019).

[4] Alfaro, L. d., and Henzinger, T. A. Interface theories for component-based design. In *International Workshop on Embedded Software* (2001), Springer, pp. 148–165.

[5] Alfaro, L. d., and Henzinger, T. A. Interface-based design. In *Engineering theories of software intensive systems*. Springer, 2005, pp. 83–104.

[6] Alur, R., Henzinger, T. A., Kupferman, O., and Vardi, M. Y. Alternating refinement relations. In *CONCUR'98 Concurrency Theory* (Berlin, Heidelberg, 1998), D. Sangiorgi and R. de Simone, Eds., Springer Berlin Heidelberg, pp. 163–178.

[7] Alur, R., Madhusudan, P., and Nam, W. Symbolic compositional verification by learning assumptions. In *Computer Aided Verification, 17th International Conference, CAV 2005, Edinburgh, Scotland, UK, July 6-10, 2005, Proceedings* (2005), pp. 548–562.

[8] Angluin, D. Learning regular sets from queries and counterexamples. *Inf. Comput. 75*, 2 (1987), 87–106.

[9] Attie, P., Baranov, E., Bliudze, S., Jaber, M., and Sifakis, J. A general framework for architecture composability. *Formal Aspects of Computing 28*, 2 (2016), 207–231.

[10] Aziz, A., Balarin, F., Brayton, R., and Sangiovanni-Vincentelli, A. L. Sequential synthesis using S1S. *IEEE Transactions on Computer-Aided Design 19*, 10 (Oct. 2000), 1149–1162.

[11] Back, R.-J., and von Wright, J. Contracts, games, and refinement. *Information and communication 156* (2000), 25–45.

[12] Bakirtzis, G., Fleming, C. H., and Vasilakopoulou, C. Categorical semantics of cyber-physical systems theory. *ACM Transactions on Cyber-Physical Systems 5*, 3 (2021), 1–32.

[13] Balarin, F., D'Angelo, M., Davare, A., Densmore, D., Meyerowitz, T., Passerone, R., Pinto, A., Sangiovanni-Vincentelli, A. L., Simalatsar, A., Watanabe, Y., Yang, G., and Zhu, Q. Chapter 10. platform-based design and frameworks. In *Model-Based Design for Embedded Systems*, G. Nicolescu and P. J. Mosterman, Eds. CRC Press, Cham, 2009, pp. 259–322.

[14] Barrett, G., and Lafortune, S. Bisimulation, the supervisory control problem and strong model matching for finite state machines. *Discrete Event Dynamic Systems: Theory & Applications 8*, 4 (Dec. 1998), 377–429.

[15] Bauer, S. S., David, A., Hennicker, R., Larsen, K. G., Legay, A., Nyman, U., and Wasowski, A. Moving from specifications to contracts in component-based design. In *Fundamental Approaches to Software Engineering: 15th International Conference, FASE 2012, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2012, Tallinn, Estonia, March 24 - April 1, 2012. Proceedings*, J. de Lara and A. Zisman, Eds. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012, pp. 43–58.

[16] Bauer, S. S., Larsen, K. G., Legay, A., Nyman, U., and Wasowski, A. A modal specification theory for components with data. *Sci. Comput. Program. 83* (2014), 106–128.

[17] Beneš, N., Delahaye, B., Fahrenberg, U., Křetínský, J., and Legay, A. Hennessy-milner logic with greatest fixed points as a complete behavioural specification theory. In *CONCUR 2013 – Concurrency Theory* (Berlin, Heidelberg, 2013), P. R. D'Argenio and H. Melgratti, Eds., Springer Berlin Heidelberg, pp. 76–90.

[18] Benveniste, A., Caillaud, B., Ferrari, A., Mangeruca, L., Passerone, R., and Sofronis, C. Multiple viewpoint contract-based specification and design. In *Formal Methods for Components and Objects, 6^{th} International Symposium (FMCO 2007), Amsterdam, The Netherlands, October 24–26, 2007, Revised Papers*, F. S. de Boer, M. M. Bonsangue, S. Graf, and Willem-Paul de Roever, Eds., vol. 5382 of *Lecture Notes in Computer Science*. Springer Verlag, Berlin Heidelberg, 2008, pp. 200–225.

[19] Benveniste, A., Caillaud, B., Nickovic, D., Passerone, R., Raclet, J.-B., Reinkemeier, P., Sangiovanni-Vincentelli, A. L., Damm, W., Henzinger, T. A., and Larsen, K. G. Contracts for system design. *Foundations and Trends® in Electronic Design Automation 12*, 2-3 (2018), 124–400.

[20] Benvenuti, L., Ferrari, A., Mangeruca, L., Mazzi, E., Passerone, R., and Sofronis, C. A contract-based formalism for the specification of heterogeneous systems. In *Proceedings of the Forum on Specification & Design Languages (FDL08)* (Stuttgart, Germany, September 23–25, 2008), pp. 142–147.

[21] Bhaduri, P., and Ramesh, S. Interface synthesis and protocol conversion. *Formal Asp. Comput. 20*, 2 (2008), 205–224.

[22] Bliudze, S., Furic, S., Sifakis, J., and Viel, A. Rigorous design of cyber-physical systems. *Software & Systems Modeling 18*, 3 (2019), 1613–1636.

[23] Bliudze, S., and Sifakis, J. Causal semantics for the algebra of connectors. *Formal methods in system design 36*, 2 (2010), 167–194.

[24] Boas, R., Cameron, B. G., and Crawley, E. F. Divergence and lifecycle offsets in product families with commonality. *Systems Engineering 16*, 2 (2013), 175–192.

[25] Bobaru, M. G., Pasareanu, C. S., and Giannakopoulou, D. Automated assume-guarantee reasoning by abstraction refinement. In *Computer Aided Verification, 20th International Conference, CAV 2008, Princeton, NJ, USA, July 7-14, 2008, Proceedings* (2008), pp. 135–148.

[26] Bochmann, G. Using logic to solve the submodule construction problem. *Discrete Event Dynamic Systems 23*, 1 (2013), 27–59.

[27] Bouyer, P., Cassez, F., and Laroussinie, F. Timed modal logics for real-time systems - specification, verification and control. *Journal of Logic, Language and Information 20*, 2 (2011), 169–203.

[28] Bujtor, F., and Vogler, W. Error-pruning in interface automata. In *40th International Conference on Current Trends in Theory and Practice of Computer Science* (Nový Smokovec, Slovakia, January 26-29, 2014), SOFSEM 2014, pp. 162–173.

[29] Burch, J., Dill, D., Wolf, E., and DeMicheli, G. Modelling hierarchical combinational circuits. In *The Proceedings of the International Conference on Computer-Aided Design* (Nov. 1993), pp. 612–617.

[30] Cancila, D., Passerone, R., Vardanega, T., and Panunzio, M. Toward correctness in the specification and handling of non-functional attributes of high-integrity real-time embedded systems. *IEEE Transactions on Industrial Informatics 6*, 2 (May 2010), 181–194.

[31] Cantor, G. Ueber eine eigenschaft des inbegriffs aller reellen algebraischen zahlen. *Journal für die reine und angewandte Mathematik 77* (1874), 258–262.

[32] Carmona, J., and Kleijn, J. Compatibility in a multi-component environment. *Theoretical Computer Science 484* (May 2013), 1–15.

[33] Cassez, F., and Laroussinie, F. Model-checking for hybrid systems by quotienting and constraints solving. In *Computer Aided Verification* (Berlin, Heidelberg, 2000), E. A. Emerson and A. P. Sistla, Eds., Springer Berlin Heidelberg, pp. 373–388.

[34] Castagnetti, G., Piccolo, M., Villa, T., Yevtushenko, N., Brayton, R. K., and Mishchenko, A. Automated synthesis of protocol converters with BALM-II. In *Software Engineering and Formal Methods. SEFM 2015 Collocated Workshops: ATSE, HOFM, MoKMaSD, and VERY*SCART. York, UK, September 7-8, 2015* (Sep 2015), D. Bianculli, R. Calinescu, and B. Rumpe, Eds., pp. 281–296.

[35] Censi, A. A mathematical theory of co-design. *CoRR abs/1512.08055* (2015).

[36] Cerny, E., and Marin, M. An approach to unified methodology of combinational switching circuits. *IEEE Transactions on Computers vol. C-26*, 8 (Aug. 1977), 745–756.

[37] Chakrabarti, A., de Alfaro, L., Henzinger, T. A., and Stoelinga, M. Resource interfaces. In *Embedded Software* (Berlin, Heidelberg, 2003), R. Alur and I. Lee, Eds., Springer Berlin Heidelberg, pp. 117–133.

[38] Chen, T., Chilton, C., Jonsson, B., and Kwiatkowska, M. A compositional specification theory for component behaviours. In *Programming Languages and Systems: 21st European Symposium on Programming, ESOP 2012, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2012, Tallinn, Estonia, March 24 - April 1, 2012. Proceedings*, H. Seidl, Ed. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012, pp. 148–168.

[39] Chen, W., Udding, J., and Verhoeff, T. Networks of communicating processes and their (de-)composition. In *Mathematics of Program Construction*, J. van de Snepscheut, Ed., vol. 375 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 1989, pp. 174–196.

[40] Chilton, C., Jonsson, B., and Kwiatkowska, M. An algebraic theory of interface automata. *Theoretical Computer Science 549* (September 2014), 146–174.

[41] Chilton, C., Jonsson, B., and Kwiatkowska, M. Compositional assume-guarantee reasoning for input/output component theories. *Science of Computer Programming 91* (2014), 115–137. Special Issue on Formal Aspects of Component Software (Selected Papers from FACS'12).

[42] Cimatti, A., and Tonetta, S. Contracts-refinement proof system for component-based embedded systems. *Science of Computer Programming 97, Part 3* (2015), 333–348.

[43] Clarke, E., Grumberg, O., Kroening, D., Peled, D., and Veith, H. *Model Checking, second edition*. Cyber Physical Systems Series. MIT Press, 2018.

[44] Clarke, E. M., and Emerson, E. A. Design and synthesis of synchronization skeletons using branching time temporal logic. In *Workshop on Logic of Programs* (1981), Springer, pp. 52–71.

[45] Clarkson, M. R., and Schneider, F. B. Hyperproperties. *Journal of Computer Security 18*, 6 (2010), 1157–1210.

[46] Cobleigh, J. M., Giannakopoulou, D., and Pasareanu, C. S. Learning assumptions for compositional verification. In *Tools and Algorithms for the Construction and Analysis of Systems, 9th International Conference, TACAS 2003, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2003, Warsaw, Poland, April 7-11, 2003, Proceedings* (2003), pp. 331–346.

[47] Coleman, J. W., and Jones, C. B. A structural proof of the soundness of rely/guarantee rules. *J. Log. Comput. 17*, 4 (2007), 807–841.

[48] Damm, W. Controlling speculative design processes using rich component models. In *Fifth International Conference on Application of Concurrency to System Design (ACSD'05)* (June 2005), pp. 118–119.

[49] Damm, W., Hungar, H., Josko, B., Peikenkamp, T., and Stierand, I. Using contract-based component specifications for virtual integration testing and architecture design. In *Design, Automation Test in Europe Conference Exhibition (DATE11)* (Grenoble, France, March 14–18, 2011), pp. 1–6.

[50] Damm, W., Votintseva, A., Metzner, A., Josko, B., Peikenkamp, T., and Böde, E. Boosting re-use of embedded automotive applications through rich components. In *Foundations of Interface Technologies* (2005), FIT'05.

[51] David, A., Larsen, K. G., Legay, A., Nyman, U., and Wasowski, A. Timed I/O automata: A complete specification theory for real-time systems. In *Proceedings of the 13th ACM International Conference on Hybrid Systems: Computation and Control* (New York, NY, USA, 2010), HSCC '10, Association for Computing Machinery, pp. 91–100.

[52] Davies, A., Brady, T., and Hobday, M. Organizing for solutions: Systems seller vs. systems integrator. *Industrial marketing management 36*, 2 (2007), 183–193.

[53] de Alfaro, L. Game models for open systems. In *Verification: Theory and Practice* (2003), N. Dershowitz, Ed., vol. 2772 of *Lecture Notes in Computer Science*, Springer Verlag, pp. 269–289.

[54] de Alfaro, L., and Henzinger, T. A. Interface automata. In *Proceedings of the 8th European Software Engineering Conference Held Jointly with 9th ACM SIGSOFT International Symposium on Foundations of Software Engineering* (New York, NY, USA, 2001), ESEC/FSE-9, Association for Computing Machinery, pp. 109–120.

[55] de Alfaro, L., and Henzinger, T. A. Interface theories for component-based design. In *EMSOFT* (2001), T. A. Henzinger and C. M. Kirsch, Eds., vol. 2211 of *Lecture Notes in Computer Science*, Springer, pp. 148–165.

[56] de Alfaro, L., Henzinger, T. A., and Stoelinga, M. Timed interfaces. In *Embedded Software* (Berlin, Heidelberg, 2002), A. Sangiovanni-Vincentelli and J. Sifakis, Eds., Springer Berlin Heidelberg, pp. 108–122.

[57] Densmore, D., and Passerone, R. A platform-based taxonomy for esl design. *IEEE Design & Test of Computers 23*, 5 (2006), 359–374.

[58] Di Benedetto, M. D., Sangiovanni-Vincentelli, A. L., and Villa, T. Model Matching for Finite State Machines. *IEEE Transactions on Automatic Control 46*, 11 (Dec. 2001), 1726–1743.

[59] Dijkstra, E. W. Solution of a problem in concurrent programming control. *Commun. ACM 8*, 9 (sep 1965), 569.

[60] Dijkstra, E. W. Guarded commands, nondeterminacy and formal derivation of programs. *Communications of the ACM 18*, 8 (August 1975), 453–457.

[61] Dill, D. L. *Trace Theory for Automatic Hierarchical Verification of Speed-Independent Circuits*. ACM Distinguished Dissertations. MIT Press, 1989.

[62] Doyen, L., Henzinger, T. A., Jobstmann, B., and Petrov, T. Interface theories with component reuse. In *Proceedings of the 8th ACM International Conference on Embedded Software* (New York, NY, USA, 2008), EMSOFT '08, Association for Computing Machinery, pp. 79–88.

[63] Dragomir, I., Ober, I., and Percebois, C. Contract-based modeling and verification of timed safety requirements within SysML. *Software & Systems Modeling* (2015), 1–38.

[64] Eilenberg, S., and MacLane, S. General theory of natural equivalences. *Transactions of the American Mathematical Society 58*, 2 (1945), 231–294.

[65] Emes, M., Smith, A., and Cowper, D. Confronting an identity crisis—how to "brand" systems engineering. *Systems Engineering 8*, 2 (2005), 164–186.

[66] Estefan, J. A., et al. Survey of model-based systems engineering (mbse) methodologies. *Incose MBSE Focus Group 25*, 8 (2007), 1–12.

[67] Ferrari, A., and Sangiovanni-Vincentelli, A. System design: Traditional concepts and new paradigms. In *Proceedings 1999 IEEE International Conference on Computer Design: VLSI in Computers and Processors (Cat. No. 99CB37040)* (1999), IEEE, pp. 2–12.

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

[69] Fujita, M., Matsunaga, Y., and Ciesielski, M. Multi-level logic optimization. In *Logic Synthesis and Verification* (2001), R. Brayton, S. Hassoun, and T. Sasao, Eds., Kluwer, pp. 29–63.

[70] Galatos, N., and Raftery, J. Idempotent residuated structures: some category equivalences and their applications. *Transactions of the American Mathematical Society 367*, 5 (2015), 3189–3223.

[71] Giannakopoulou, D., Pasareanu, C. S., and Barringer, H. Assumption generation for software component verification. In *17th IEEE International Conference on Automated Software Engineering (ASE 2002), 23-27 September 2002, Edinburgh, Scotland, UK* (2002), pp. 3–12.

[72] Goguen, J. A., and Meseguer, J. Security policies and security models. In *1982 IEEE Symposium on Security and Privacy, Oakland, CA, USA, April 26-28, 1982* (Oakland, CA, USA, 1982), IEEE Computer Society, pp. 11–20.

[73] Golan, J. S. *Semirings and their Applications*, 1st ed ed. Springer, Dordrecht, 1999.

[74] Graf, S., Passerone, R., and Quinton, S. Contract-based reasoning for component systems with rich interactions. In *Embedded Systems Development: From Functional Models to Implementations*, A. L. Sangiovanni-Vincentelli, H. Zeng, M. D. Natale, and P. Marwedel, Eds., vol. 20 of *Embedded Systems*. Springer New York, 2014, ch. 8, pp. 139–154.

[75] Green, P. Protocol conversion. *IEEE Transactions on Communications 34*, 3 (Mar 1986), 257–268.

[76] Griffin, M. D. How do we fix systems engineering? In *61st international astronautical congress* (2010), vol. 27.

[77] Group, O. M. Object constraint language, version 2.0, May 2006.

[78] Haghverdi, E., and Ural, H. Submodule construction from concurrent system specifications. *Information and Software Technology 41*, 8 (June 1999), 499–506.

[79] Hallal, H., Negulescu, R., and Petrenko, A. Design of divergence-free protocol converters using supervisory control techniques. In *7th IEEE International Conference on Electronics, Circuits and Systems, ICECS 2000* (Dec. 2000), vol. 2, pp. 705–708.

[80] Hansen, S., Berente, N., and Lyytinen, K. Requirements in the 21st century: Current practice and emerging trends. In *Design Requirements Engineering: A Ten-Year Perspective* (Berlin, Heidelberg, 2009), K. Lyytinen, P. Loucopoulos, J. Mylopoulos, and B. Robinson, Eds., Springer Berlin Heidelberg, pp. 44–87.

[81] Harel, D., and Pnueli, A. On the development of reactive systems. In *Logics and Models of Concurrent Systems* (Berlin, Heidelberg, 1985), K. R. Apt, Ed., Springer Berlin Heidelberg, pp. 477–498.

[82] Hassoun, S., and Villa, T. Optimization of synchronous circuits. In *Logic Synthesis and Verification* (2001), R. Brayton, S. Hassoun, and T. Sasao, Eds., Kluwer, pp. 225–253.

[83] Hayes, I. J., and Jones, C. B. A guide to rely/guarantee thinking. In *Engineering Trustworthy Software Systems - Third International School, SETSS 2017, Chongqing, China, April 17-22, 2017, Tutorial Lectures* (2017), J. P. Bowen, Z. Liu, and Z. Zhang, Eds., vol. 11174 of *Lecture Notes in Computer Science*, Springer, pp. 1–38.

[84] Henzinger, T. A., Jhala, R., and Majumdar, R. Permissive interfaces. *SIGSOFT Softw. Eng. Notes 30*, 5 (sep 2005), 31–40.

[85] Henzinger, T. A., and Sifakis, J. The discipline of embedded systems design. *Computer 40*, 10 (2007), 32–40.

[86] Hitchins, D. K. *Putting systems to work*, vol. 325. Wiley Chichester, 1992.

[87] Hoare, C. A. R. An axiomatic basis for computer programming. *Commun. ACM 12*, 10 (Oct. 1969), 576–580.

[88] Hoare, C. A. R. A model for communicating sequential process. Tech. Rep. PRG-22, Oxford University, Programming Research Group, 1981.

[89] Hobday, M., Davies, A., and Prencipe, A. Systems integration: a core capability of the modern corporation. *Industrial and corporate change 14*, 6 (2005), 1109–1143.

[90] Hopcroft, J., Motwani, R., and Ullman, J. *Introduction to Automata Theory, Languages, and Computation.* Addison-Wesley Publishing Company, 2001.

[91] Iannopollo, A. *A Platform-Based Approach to Verification and Synthesis of Linear Temporal Logic Specifications.* PhD thesis, UC Berkeley, 2018.

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

[93] Incer, I., Benveniste, A., Sangiovanni-Vincentelli, A. L., and Seshia, S. A. Hypercontracts. *arXiv preprint arXiv:2106.02449* (2021).

[94] Incer, I., Benveniste, A., Sangiovanni-Vincentelli, A. L., and Seshia, S. A. Hypercontracts. In *NASA Formal Methods* (Cham, 2022), J. Deshmukh, K. Havelund, and I. Perez, Eds., Springer International Publishing.

[95] Incer, I., Mangeruca, L., Villa, T., and Sangiovanni-Vincentelli, A. L. The quotient in preorder theories. In *Proceedings 11th International Symposium on Games, Automata, Logics, and Formal Verification, Brussels, Belgium, September 21-22, 2020* (Brussels, Belgium, 2020), J.-F. Raskin and D. Bresolin, Eds., vol. 326 of *Electronic Proceedings in Theoretical Computer Science*, Open Publishing Association, pp. 216–233.

[96] Incer, I., Sangiovanni-Vincentelli, A. L., Lin, C.-W., and Kang, E. Quotient for assume-guarantee contracts. In *16th ACM-IEEE International Conference on Formal Methods and Models for System Design* (October 2018), MEMOCODE'18, pp. 67–77.

[97] Jackson, S. Memo to industry: The crisis in systems engineering. *INSIGHT 13*, 1 (2010), 43–43.

[98] Jensen, J. C., Chang, D. H., and Lee, E. A. A model-based design methodology for cyber-physical systems. In *2011 7th international wireless communications and mobile computing conference* (2011), IEEE, pp. 1666–1671.

[99] Jha, S., and Seshia, S. A. A Theory of Formal Synthesis via Inductive Learning. *Acta Informatica 54*, 7 (2017), 693–726.

[100] Jones, C. B. Specification and design of (parallel) programs. In *IFIP Congress* (Paris, France, 1983), pp. 321–332.

[101] Jones, C. B. Wanted: a compositional approach to concurrency. In *Programming Methodology* (New York, NY, 2003), A. McIver and C. Morgan, Eds., Springer New York, pp. 5–15.

[102] Kam, T., Villa, T., Brayton, R. K., and Sangiovanni-Vincentelli, A. L. *Synthesis of FSMs: Functional Optimization*. Kluwer Academic Publishers, Boston, 1997.

[103] Karsai, G., Sztipanovitz, J., Ledczi, A., and Bapty, T. Model-integrated development of embedded software. *Proceedings of the IEEE 91*, 1 (January 2003).

[104] Keller, R. M. Formal verification of parallel programs. *Commun. ACM 19*, 7 (July 1976), 371–384.

[105] Keutzer, K., Newton, A. R., Rabaey, J. M., and Sangiovanni-Vincentelli, A. System-level design: Orthogonalization of concerns and platform-based design. *IEEE transactions on computer-aided design of integrated circuits and systems 19*, 12 (2000), 1523–1543.

[106] Kim, E. S., Arcak, M., and Seshia, S. A. A small gain theorem for parametric assume-guarantee contracts. In *Proceedings of the 20th International Conference on Hybrid Systems: Computation and Control* (New York, NY, USA, 2017), HSCC '17, Association for Computing Machinery, pp. 207–216.

[107] Kim, J., and Newborn, M. The simplification of sequential machines with input restrictions. *IRE Transactions on Electronic Computers* (Dec. 1972), 1440–1443.

[108] Kovalyov, S. P. Methods of the category theory in digital design of heterogeneous cyber-physical systems. *Informatika i Ee Primeneniya [Informatics and its Applications] 15*, 1 (2021), 23–29.

[109] Koymans, R. Specifying real-time properties with metric temporal logic. *Real-Time Systems 2*, 4 (Nov 1990), 255–299.

[110] Kumar, R., Nelvagal, S., and Marcus, S. A discrete event systems approach for protocol conversion. *Discrete Event Dynamic Systems: Theory & Applications 7*, 3 (June 1997), 295–315.

[111] Kunnummel, V. A conceptual modelling framework for evaluation of cyber-physical systems based on applied category theory and metamodeling. In *PoEM Doctoral Consortium* (2018), pp. 74–85.

[112] Kurshan, R., Merritt, M., Orda, A., and Sachs, S. Modelling asynchrony with a synchronous model. *Formal Methods in System Design vol. 15*, no. 3 (Nov. 1999), 175–199.

[113] Lam, S. S. Protocol conversion. *IEEE Trans. Softw. Eng. 14*, 3 (Mar. 1988), 353–362.

[114] Lamport, L. What it means for a concurrent program to satisfy a specification: Why no one has specified priority. In *Proceedings of the 12th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages* (New York, NY, USA, 1985), POPL '85, Association for Computing Machinery, pp. 78–83.

[115] Lamport, L. win and sin: Predicate transformers for concurrency. *ACM Transactions on Programming Languages and Systems 12*, 3 (July 1990), 396–428.

[116] Lamport, L. The computer science of concurrency: The early years. *Commun. ACM 58*, 6 (may 2015), 71–76.

[117] Larsen, K., and Xinxin, L. Equation solving using modal transition systems. In *Logic in Computer Science, 1990. LICS '90, Proceedings., Fifth Annual IEEE Symposium on e* (Jun 1990), pp. 108–117.

[118] Larsen, K. G., Nyman, U., and Wasowski, A. Interface Input/Output Automata. In *FM* (2006), pp. 82–97.

[119] Larsen, K. G., Nyman, U., and Wasowski, A. Modal I/O Automata for Interface and Product Line Theories. In *Programming Languages and Systems, 16th European Symposium on Programming, ESOP'07* (2007), vol. 4421 of *Lecture Notes in Computer Science*, Springer, pp. 64–79.

[120] Larsen, K. G., Nyman, U., and Wasowski, A. On Modal Refinement and Consistency. In *Proc. of the 18th International Conference on Concurrency Theory (CONCUR'07)* (2007), Springer, pp. 105–119.

[121] Le, T. T. H., Passerone, R., Fahrenberg, U., and Legay, A. Contract-based requirement modularization via synthesis of correct decompositions. *ACM Trans. Embed. Comput. Syst. 15*, 2 (Feb. 2016), 33:1–33:26.

[122] Ledeczi, A., Bakay, A., Maroti, M., Volgyesi, P., Nordstrom, G., Sprinkle, J., and Karsai, G. Composing domain-specific design environments. *IEEE Computer 34*, 11 (November 2001), 44 –51.

[123] Lee, E. A. Cyber-physical systems-are computing foundations adequate. In *Position paper for NSF workshop on cyber-physical systems: research motivation, techniques and roadmap* (2006), vol. 2, pp. 1–9.

[124] Lee, E. A. Cyber physical systems: Design challenges. In *2008 11th IEEE international symposium on object and component-oriented real-time distributed computing (ISORC)* (2008), IEEE, pp. 363–369.

[125] Lee, E. A. Constructive models of discrete and continuous physical phenomena. *IEEE Access 2* (2014), 797–821.

[126] Lee, E. A. The past, present and future of cyber-physical systems: A focus on models. *Sensors 15*, 3 (2015), 4837–4869.

[127] Lee, E. A. Fundamental limits of cyber-physical systems modeling. *ACM Transactions on Cyber-Physical Systems 1*, 1 (2016), 1–26.

[128] Lee, E. A., and Sangiovanni-Vincentelli, A. A framework for comparing models of computation. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 17*, 12 (Dec 1998), 1217–1229.

[129] Lee, E. A., and Sangiovanni-Vincentelli, A. L. Component-based design for the future. In *2011 Design, Automation & Test in Europe* (2011), IEEE, pp. 1–5.

[130] Lee, E. A., and Seshia, S. A. *Introduction to embedded systems: A cyber-physical systems approach*. MIT Press, 2016.

[131] Legatiuk, D., Theiler, M., Dragos, K., and Smarsly, K. A categorical approach towards metamodeling cyber-physical systems. In *Proceedings of the 11th International Workshop on Structural Health Monitoring (IWSHM). Stanford, CA, USA* (2017), vol. 9.

[132] Leveson, N. A new accident model for engineering safer systems. *Safety Science 42*, 4 (2004), 237–270.

[133] Leveson, N. G. Software safety: Why, what, and how. *ACM Comput. Surv. 18*, 2 (jun 1986), 125–163.

[134] Liskov, B. H., and Wing, J. M. A behavioral notion of subtyping. *ACM Transactions on Programming Languages and Systems 16*, 6 (November 1994), 1811–1841.

[135] Lüttgen, G., and Vogler, W. Modal interface automata. *Logical Methods in Computer Science 9*, 3 (2013).

[136] Lynch, N. A., and Tuttle, M. R. An introduction to input/output automata. *CWI Quarterly 2* (1989), 219–246.

[137] Mac Lane, S. Concepts and categories in perspective. *Duren P, A century of mathematics in America Part 3* (1988), 353–365.

[138] Mac Lane, S. *Categories for the working mathematician. 2nd ed.*, 2nd ed ed., vol. 5. New York, NY: Springer, 1998.

[139] Madni, A. M. Adaptable platform-based engineering: Key enablers and outlook for the future. *Systems Engineering 15*, 1 (2012), 95–107.

[140] Madni, A. M. Elegant systems design: Creative fusion of simplicity and power. *Systems Engineering 15*, 3 (2012), 347–354.

[141] Madni, A. M., and Sievers, M. Systems integration: Key perspectives, experiences, and challenges. *Systems Engineering 17*, 1 (2014), 37–51.

[142] Maler, O., and Nickovic, D. Monitoring temporal properties of continuous signals. In *Formal Techniques, Modelling and Analysis of Timed and Fault-Tolerant Systems* (Berlin, Heidelberg, 2004), Y. Lakhnech and S. Yovine, Eds., Springer Berlin Heidelberg, pp. 152–166.

[143] Mallon, W., Tijmen, J., and Verhoeff, T. Analysis and applications of the XDI model. In *International Symposium on Advanced Research in Asynchronous Circuits and Systems* (1999), pp. 231–242.

[144] Mallozzi, P., Nuzzo, P., and Pelliccione, P. Incremental refinement of goal models with contracts. In *Fundamentals of Software Engineering* (Cham, 2021), H. Hojjat and M. Massink, Eds., Springer International Publishing, pp. 35–50.

[145] Mangeruca, L., Ferrante, O., and Ferrari, A. Formalization and completeness of evolving requirements using contracts. In *Proceedings of the* 8^{th} *IEEE International Symposium on Industrial Embedded Systems* (Porto, Portugal, June 19–21 2013), SIES 2013, pp. 120–129.

[146] Mastroeni, I., and Pasqua, M. Verifying bounded subset-closed hyperproperties. In *Static Analysis* (Cham, 2018), A. Podelski, Ed., Springer International Publishing, pp. 263–283.

[147] Merlin, P., and v. Bochmann, G. On the construction of submodule specifications and communication protocols. *ACM Transactions on Programming Languages and Systems 5*, 1 (Jan. 1983), 1–25.

[148] Meyer, B. Applying "design by contract". *IEEE Computer 25*, 10 (October 1992), 40–51.

[149] Meyer, B. *Touch of Class: Learning to Program Well Using Object Technology and Design by Contracts*. Springer, Software Engineering, 2009.

[150] Mordecai, Y., Fairbanks, J., and Crawley, E. F. Category-theoretic formulation of the model-based systems architecting cognitive-computational cycle. *Applied Sciences 11*, 4 (2021), 1945.

[151] National Defense Industrial Association and others. Top systems engineering issues in US defense industry. *Systems Engineering Division Task Group Report* (2016).

[152] Neches, R., and Madni, A. M. Towards affordably adaptable and effective systems. *Systems Engineering 16*, 2 (2013), 224–234.

[153] Negulescu, R. Process spacess. Tech. Rep. CS-95-48, University of Waterloo, 1995.

[154] Negulescu, R. Process spaces. In *Proceedings of CONCUR 2000, 11th International Conference on Concurrency Theory* (2000), C. Palamidessi, Ed., vol. 1877 of *LNCS*, Springer-Verlag, pp. 199–213.

[155] Nuzzo, P. *Compositional design of cyber-physical systems using contracts*. PhD thesis, UC Berkeley, 2015.

[156] Nuzzo, P., Sangiovanni-Vincentelli, A. L., Bresolin, D., Geretti, L., and Villa, T. A platform-based design methodology with contracts and related tools for the design of cyber-physical systems. *Proceedings of the IEEE 103*, 11 (2015), 2104–2132.

[157] Nuzzo, P., Xu, H., Ozay, N., Finn, J. B., Sangiovanni-Vincentelli, A. L., Murray, R. M., Donzé, A., and Seshia, S. A. A contract-based methodology for aircraft electric power system design. *IEEE Access 2* (2014), 1–25.

[158] Oster, C., and Wade, J. Ecosystem requirements for composability and reuse: An investigation into ecosystem factors that support adoption of composable practices for engineering design. *Systems Engineering 16*, 4 (2013), 439–452.

[159] Parnas, D. L. A technique for software module specification with examples. *Commun. ACM 15*, 5 (may 1972), 330–336.

[160] Passerone, R., de Alfaro, L., Henzinger, T. A., and Sangiovanni-Vincentelli, A. L. Convertibility verification and converter synthesis: two faces of the same coin. In *ICCAD* (2002), L. T. Pileggi and A. Kuehlmann, Eds., ACM, pp. 132–139.

[161] Passerone, R., Incer, I., and Sangiovanni-Vincentelli, A. L. Coherent extension, composition, and merging operators in contract models for system design. *ACM Trans. Embed. Comput. Syst. 18*, 5s (Oct. 2019).

[162] Passerone, R., Incer, I., and Sangiovanni-Vincentelli, A. L. Contract model operators for composition and merging: extensions and proofs. Technical Report DISI-19-004, Dipartimento di Ingegneria e Scienza dell'Informazione, University of Trento, August 2019.

[163] Passerone, R., Rowson, J. A., and Sangiovanni-Vincentelli, A. L. Automatic synthesis of interfaces between incompatible protocols. In *DAC* (1998), pp. 8–13.

[164] Phan-Minh, T. *Contract-Based Design: Theories and Applications*. PhD thesis, California Institute of Technology, 2021.

[165] Phan-Minh, T., and Murray, R. M. Contracts of reactivity. Tech. rep., California Institute of Technology, 2019.

[166] Pnueli, A. The temporal logic of programs. In *18th Annual Symposium on Foundations of Computer Science (sfcs 1977)(FOCS)* (Sept 1977), pp. 46–57.

[167] Rabe, M. N. *A temporal logic approach to information-flow control*. PhD thesis, Universität des Saarlandes, 2016.

[168] Raclet, J. Residual for component specifications. *Electr. Notes Theor. Comput. Sci. 215* (2008), 93–110.

[169] Raclet, J.-B., Badouel, E., Benveniste, A., Caillaud, B., Legay, A., and Passerone, R. Modal interfaces: Unifying interface automata and modal specifications. In *Proceedings of the Seventh ACM International Conference on Embedded Software* (New York, NY, USA, 2009), EMSOFT '09, Association for Computing Machinery, pp. 87–96.

[170] Raclet, J.-B., Badouel, E., Benveniste, A., Caillaud, B., Legay, A., and Passerone, R. A modal interface theory for component-based design. *Fundamenta Informaticae 108*, 1–2 (2011), 119–149.

[171] Ramos, A. L., Ferreira, J. V., and Barceló, J. Model-based systems engineering: An emerging approach for modern systems. *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews) 42*, 1 (2011), 101–111.

[172] Rowson, J. A., and Sangiovanni-Vincentelli, A. Interface-based design. In *Proceedings of the 34th annual Design Automation Conference* (1997), pp. 178–183.

[173] Sage, A. P., and Lynch, C. L. Systems integration and architecting: An overview of principles, practices, and perspectives. *Systems Engineering: The Journal of The International Council on Systems Engineering 1*, 3 (1998), 176–227.

[174] Sangiovanni-Vincentelli, A. Quo vadis, SLD? Reasoning about the trends and challenges of system level design. *Proceedings of the IEEE 95*, 3 (March 2007), 467–506.

[175] Sangiovanni-Vincentelli, A., Carloni, L., De Bernardinis, F., and Sgroi, M. Benefits and challenges for platform-based design. In *Proceedings of the 41st Annual Design Automation Conference* (2004), pp. 409–414.

[176] Sangiovanni-Vincentelli, A., and Martin, G. Platform-based design and software design methodology for embedded systems. *IEEE Design & Test of computers 18*, 6 (2001), 23–33.

[177] Sangiovanni-Vincentelli, A. L., Damm, W., and Passerone, R. Taming Dr. Frankenstein: Contract-Based Design for Cyber-Physical Systems. *European Journal of Control 18*, 3 (2012), 217 – 238.

[178] Saoud, A., Girard, A., and Fribourg, L. On the composition of discrete and continuous-time assume-guarantee contracts for invariance. In *16th European Control Conference, ECC, June 12-15, 2018* (Limassol, Cyprus, 2018), IEEE, pp. 435–440.

[179] Saoud, A., Girard, A., and Fribourg, L. Assume-guarantee contracts for continuous-time systems. working paper or preprint, Feb. 2021.

[180] Schmidt, D. Model-driven engineering. *IEEE Computer* (February 2006), 25–31.

[181] Sentovich, E., and Brand, D. Flexibility in logic. In *Logic Synthesis and Verification* (2001), R. Brayton, S. Hassoun, and T. Sasao, Eds., Kluwer, pp. 65–88.

[182] Seshia, S. A. Combining induction, deduction, and structure for verification and synthesis. *Proceedings of the IEEE 103*, 11 (2015), 2036–2051.

[183] Seshia, S. A., Desai, A., Dreossi, T., Fremont, D. J., Ghosh, S., Kim, E., Shivakumar, S., Vazquez-Chanlatte, M., and Yue, X. Formal specification for deep neural networks. In *Automated Technology for Verification and Analysis* (Cham, 2018), S. K. Lahiri and C. Wang, Eds., Springer International Publishing, pp. 20–34.

[184] Shali, B. M., van der Schaft, A. J., and Besselink, B. Behavioural assume-guarantee contracts for linear dynamical systems, 2021.

[185] Shallcross, N., Parnell, G. S., Pohl, E., and Specking, E. Set-based design: The state-of-practice and research opportunities. *Systems Engineering 23*, 5 (2020), 557–578.

[186] Shannon, C. E., and McCarthy, J., Eds. *Automata studies*, vol. 11. Princeton University Press Princeton, 1956.

[187] Sifakis, J. Rigorous system design. *Foundations and Trends® in Electronic Design Automation 6*, 4 (2013), 293–362.

[188] Sifakis, J. Toward a system design science. In *From Programs to Systems. The Systems perspective in Computing*. Springer, 2014, pp. 225–234.

[189] Sifakis, J. System design automation: Challenges and limitations. *Proceedings of the IEEE 103*, 11 (2015), 2093–2103.

[190] Sifakis, J., Bensalem, S., Bliudze, S., and Bozga, M. A theory agenda for component-based design. In *Software, Services, and Systems*. Springer, 2015, pp. 409–439.

[191] Singer, D. J., Doerry, N., and Buckley, M. E. What is set-based design? *Naval Engineers Journal 121*, 4 (Oct 2009), 31–43.

[192] Speranzon, A., Spivak, D. I., and Varadarajan, S. Abstraction, composition and contracts: A sheaf theoretic approach, 2018.

[193] Tripakis, S., Lickly, B., Henzinger, T. A., and Lee, E. A. A theory of synchronous relational interfaces. *ACM Transactions on Programming Languages and Systems 33*, 4 (July 2011).

[194] Tripakis, S., Stergiou, C., Broy, M., and Lee, E. A. Error-completion in interface theories. In *Model Checking Software*, E. Bartocci and C. Ramakrishnan, Eds., vol. 7976 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2013, pp. 358–375.

[195] Turing, A. M. On checking a large routine. In *Report of a Conference on High-Speed Automatic Calculating Machines* (Cambridge, 1949), University Mathematical Laboratory, pp. 67–69.

[196] Villa, T., Petrenko, A., Yevtushenko, N., Mishchenko, A., and Brayton, R. K. Component-based design by solving language equations. *Proceedings of the IEEE 103*, 11 (Nov 2015), 2152–2167.

[197] Villa, T., Yevtushenko, N., Brayton, R., Mishchenko, A., Petrenko, A., and Sangiovanni-Vincentelli, A. *The Unknown Component Problem: Theory and Applications*. Springer, 2012.

[198] Warmer, J., and Kleppe, A. *The Object Constraint Language: Getting Your Models Ready for MDA*, 2nd ed. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.

[199] Watanabe, S., Seto, K., Ishikawa, Y., Komatsu, S., and Fujita, M. Protocol transducer synthesis using divide and conquer approach. In *Design Automation Conference, 2007. ASP-DAC '07. Asia and South Pacific* (January 2007), pp. 280–285.

[200] Watson, M. D., Griffin, M., Farrington, P. A., Burns, L., Colley, W., Collopy, P., Doty, J., Johnson, S. B., Malak, R., Shelton, J., et al. Building a path to elegant design. In *Proceedings of the International Annual Conference of the American Society for Engineering Management.* (2014), American Society for Engineering Management (ASEM), p. 1.

[201] Willems, J. C. *System Theoretic Foundations for Modelling Physical Systems*. Springer Vienna, Vienna, 1980, pp. 279–289.

[202] Willems, J. C. The behavioral approach to open and interconnected systems. *IEEE Control Systems Magazine 27*, 6 (Dec 2007), 46–99.

[203] Wolf, E. S. *Hierarchical Models of Synchronous Circuits for Formal Verification and Substitution*. PhD thesis, Department of Computer Science, Stanford University, October 1995.

[204] Wymore, A. W. *Model-based systems engineering*. CRC press, 1993.

[205] Wymore, A. W. Systems movement: Autobiographical retrospectives. *International Journal of General Systems 33*, 6 (2004), 593–610.

[206] Yevtushenko, N., Villa, T., Brayton, R. K., Mishchenko, A., and Sangiovanni-Vincentelli, A. L. Composition operators in language equations. In *International Workshop on Logic and Synthesis* (June 2004), pp. 409–415.

[207] Zardini, G., Spivak, D. I., Censi, A., and Frazzoli, E. A compositional sheaf-theoretic framework for event-based systems. *Electronic Proceedings in Theoretical Computer Science 333* (feb 2021), 139–153.

Appendix A

Additional proofs

A.1 Receptive languages and hypercontracts

Proof of Proposition 8.5.2. Suppose $L, L' \in \mathcal{L}_I$. If w is contained in $L \cap L'$, and w_p is a prefix of w, then w is contained in both L and L', and so is w_p, which means intersection is prefix-closed. Moreover, for any $w' \in I^*$, we have $w \circ w' \in L$ and $w \circ w' \in L'$, so $w \circ w' \in L \cap L'$. We conclude that $L \cap L' \in \mathcal{L}_I$.

Similarly, if w is contained in $L \cup L'$, then we may assume that $w \in L$. Any prefix w_p of w is also contained in L, so $w_p \in L \cup L'$, meaning that union is prefix-closed. In addition, for every $w' \in I^*$, we have $w \circ w' \in L$, so $w \circ w' \in L \cup L'$. This means that $L \cup L' \in \mathcal{L}_I$. □

Proof of Proposition 8.5.3. First we show that $L' \to L \in \mathcal{L}_I$. Let $w \in L' \to L$. If w_p is a prefix of w then $\mathrm{Pre}(w_p) \cap L' \subseteq \mathrm{Pre}(w) \cap L' \subseteq L$, so $L' \to L$ is prefix-closed.

Now suppose $w \in L' \to L$ and $w \in L$. Then for $w_I \in I^*$, $w \circ w_I \in L$, so $\mathrm{Pre}(w \circ w_I) \subseteq L$. Suppose $w \in L' \to L$ and $w \notin L$. Let n be the length of w. Since $w \notin L$, $n > 0$ (the empty string is in L). Write $w = \sigma_1 \dots \sigma_n$ for $\sigma_i \in \Sigma$. Let $k \leq n$ be the largest natural number such that $\sigma_1 \dots \sigma_k \in L'$ (note that k can be zero). If $k = n$, then $w \in L' \cap \mathrm{Pre}(w) \subseteq L$, which is forbidden by our assumption that $w \notin L$. Thus, $k < n$. Define $w_p = \sigma_1 \dots \sigma_{k+1}$. Clearly, $w_p \notin L'$. For any $w_\Sigma \in \Sigma^*$, since L' is prefix-closed, we must have $\mathrm{Pre}(w \circ w_\Sigma) \cap L' = \mathrm{Pre}(w_p) \cap L' = \mathrm{Pre}(w) \cap L' \subseteq L$. We showed that any word of $L' \to L$ extended by a word of I^* remains in $L' \to L$. We conclude that $L' \to L \in \mathcal{L}_I$.

Now we show that $L' \to L$ has the properties of the exponential. Suppose $L'' \in \mathcal{L}_I$ is such that $L' \cap L'' \subseteq L$. Let $w \in L''$. Then $\mathrm{Pre}(w) \cap L' \subseteq L$, which means that $L'' \leq L' \to L$. On the other hand,

$$L' \cap (L' \to L) = L' \cap \{w \in \Sigma^* \mid \mathrm{Pre}(w) \cap L' \subseteq L\} \subseteq L.$$

Thus, any $L'' \leq L' \to L$ satisfies $L'' \cap L' \leq L$. This concludes the proof. □

Proof of Proposition 8.5.4. From Prop. 8.5.3, it is clear that $L \subseteq L' \to L$. Suppose $w \in L' \cap L$. Since L and L' are I-receptive, $w \circ \sigma \in L \cap L'$ for $\sigma \in I$. Assume $\sigma \in O$. If $w \circ \sigma \notin L'$,

then we can extend $w \circ \sigma$ by any word $w' \in \Sigma^*$, and this will satisfy $\text{Pre}(w \circ \sigma \circ w') \cap L' = \text{Pre}(w) \cap L' \subseteq L$ due to the fact L' is prefix-closed. If $w \circ \sigma \in L' - L$, then $w \notin L' \to L$. Thus, we can express the exponential using the closed-form expression of the proposition. □

Proof of Proposition 8.5.6. Suppose $w \in L/L'$ and $w \in L \cap L'$. We have not lost generality because $\epsilon \in L \cap L'$. We consider extensions of w by a symbol σ:

a. If $\sigma \in I$, σ is an input symbol for both L' and the quotient.
   i. L is receptive to I, so $w \circ \sigma \in L$;
   ii. L' is receptive to $I \subseteq I'$, so $w \circ \sigma \in L'$; and
   iii. L/L' must contain $w \circ \sigma$ because the quotient is I_r-receptive.

b. If $\sigma \in O \cap I'$, then σ is an output of the quotient, and an input of L'.
   i. L' is I'-receptive, so $w \circ \sigma \in L'$;
   ii. σ is an output symbol for both L and L/L', so none of them is required to contain $w \circ \sigma$; and
   iii. if $w \circ \sigma \in L' - L$, the extension $w \circ \sigma$ cannot be in the quotient. Otherwise, it can.

c. If $\sigma \in O'$, σ is an output for L' and an input for the quotient.
   i. Neither L nor L' are O'-receptive;
   ii. L/L' is O'-receptive, so we must have $w \circ \sigma \in L/L'$; and
   iii. if $w \circ \sigma \in L' - L$, we cannot have $w \circ \sigma \in L/L'$.

Starting with a word w in the quotient, statements a and b allow or disallow extensions of that word to be in the quotient. However, statements c.ii and c.iii impose a requirement on the word w itself, i.e., if c.iii is violated, c.ii implies that w is not in the quotient. Statements a.iii and c.ii impose an obligation on the quotient to accept extensions by symbols of I and O'; and those extensions may lead to a violation of c.iii. Thus, we remove from the quotient all words such that extensions of those words by elements of $I \cup O'$ end up in $L' - L$. The expression of the proposition follows from these considerations. □

Proof of Proposition 8.5.7. From the principle of hypercontract composition, we must have

$$E_R \leq U \stackrel{\text{def}}{=} (E_{S'}/M_S) \wedge (E_S/M_{S'}) \text{ and} \tag{A.1}$$

$$L \stackrel{\text{def}}{=} M_{S'} \times M_S \leq M_R. \tag{A.2}$$

Observe that the quotients $E_{S'}/M_S$ and $E_S/M_{S'}$ both have IO signature $O \cup O'$, so the conjunction in (A.1) is well-defined as an operation of the Heyting algebra $\mathcal{L}_{O \cup O'}$. We study the first element:

$$E_{S'}/M_S = (E_{S'} \cap M_S \cup \mathrm{MissExt}(E_{S'}, M_S, O)) - \mathrm{Unc}(E_{S'}, M_S, O, O').$$

We attempt to simplify the terms. Suppose $w \in E_{S'} \cap (M_S - S)$. Then all extensions of w lie in $M_S - S$. This means that $\mathrm{MissExt}(E_{S'}, M_S, O) = \mathrm{MissExt}(E_{S'}, S, O)$. Moreover, if a word is an element of $E_{S'} - S'$, all its extensions are in this set, as well (i.e., it is impossible to escape this set by extending words). Thus, $\mathrm{Unc}(E_{S'}, M_S, O, O') = \mathrm{Unc}(S', M_S, O, O')$. We have

$$E_{S'}/M_S = (E_{S'} \cap M_S \cup \mathrm{MissExt}(E_{S'}, S, O)) - \mathrm{Unc}(S', M_S, O, O').$$

Now we can write

$$U = \begin{bmatrix} (E_{S'} \cap M_S \cup \mathrm{MissExt}(E_{S'}, S, O)) \cap \\ (E_S \cap M_{S'} \cup \mathrm{MissExt}(E_S, S', O')) \end{bmatrix} - [\mathrm{Unc}(S', M_S, O, O') \cup \mathrm{Unc}(S, M_{S'}, O', O)].$$

Observe that

$$\begin{aligned} &E_{S'} \cap M_S \cap \mathrm{MissExt}(E_S, S', O') \\ &= (S' \cup \mathrm{MissExt}(S', S', O')) \cap M_S \cap \mathrm{MissExt}(E_S, S', O') \\ &= M_S \cap \mathrm{MissExt}(E_S, S', O') = M_S \cap \mathrm{MissExt}(S, S', O'). \end{aligned}$$

The last equality comes from the following fact: if a word of $\mathrm{MissExt}(E_S, S', O')$ is obtained by extending a word of $(E_S - S) \cap S'$ by O', the resulting word is still an element of E_S, which means it cannot be an element of M_S because M_S and E_S are disjoint outside of S. Therefore,

$$U = \begin{bmatrix} (S \cap S') \cup \\ (M_S \cap \mathrm{MissExt}(S, S', O')) \cup \\ (M_{S'} \cap \mathrm{MissExt}(S', S, O)) \cup \\ (\mathrm{MissExt}(E_{S'}, S, O) \cap \mathrm{MissExt}(E_S, S', O')) \end{bmatrix} - [\mathrm{Unc}(S', M_S, O, O') \cup \mathrm{Unc}(S, M_{S'}, O', O)]. \tag{A.3}$$

We can write

$$\begin{aligned} \mathrm{MissExt}(E_{S'}, S, O) \cap \mathrm{MissExt}(E_S, S', O') = \\ \mathrm{MissExt}(E_{S'}, S, O) \cap \mathrm{MissExt}(S, S', O') \cup \\ \mathrm{MissExt}(S', S, O) \cap \mathrm{MissExt}(E_S, S', O') \cup \\ \begin{pmatrix} \mathrm{MissExt}(\mathrm{MissExt}(S', S', O'), S, O) \cap \\ \mathrm{MissExt}(\mathrm{MissExt}(S, S, O), S', O') \end{pmatrix}. \end{aligned}$$

Note that $\text{MissExt}(E_{S'}, S, O) \cap \text{MissExt}(S, S', O') = \text{MissExt}(S, S, O) \cap \text{MissExt}(S, S', O')$. Hence

$$\begin{aligned}
&(M_S \cap \text{MissExt}(S, S', O')) \cup \\
&(\text{MissExt}(E_{S'}, S, O) \cap \text{MissExt}(S, S', O')) \\
&= \text{MissExt}(S, S', O') \cap (M_S \cup \text{MissExt}(E_{S'}, S, O)) \\
&= \text{MissExt}(S, S', O') \cap (M_S \cup \text{MissExt}(S, S, O)) \\
&= \text{MissExt}(S, S', O').
\end{aligned}$$

Finally, we observe that the set

$$\text{MissExt}(\text{MissExt}(S', S', O'), S, O) \cap \text{MissExt}(\text{MissExt}(S, S, O), S', O')$$

must be empty since the words of the first term have prefixes in $S - S'$, and the second in $S' - S$. These considerations allow us to conclude that

$$\begin{aligned}
U \leq \; &\begin{bmatrix} (S \cap S') \cup \\ \text{MissExt}(S, S', O') \cup \\ \text{MissExt}(S', S, O). \end{bmatrix} - \\
&\big[\text{Unc}(S', M_S, O, O') \cup \text{Unc}(S, M_{S'}, O', O)\big].
\end{aligned}$$

To simplify the expression a step further, suppose $w \in \text{Unc}(S', M_S, O, O')$ and has a prefix in $S' \cap (M_S - S)$. Then $w \notin S \cap S'$. The words of $\text{MissExt}(S, S', O')$ do not have prefixes in $S' - S$, so $w \notin \text{MissExt}(S, S', O')$. The words of $\text{MissExt}(S', S, O)$ belong to E_S, which is disjoint from M_S outside of S. Thus, $w \notin \text{MissExt}(S', S, O)$.

We just learned that the words of $\text{Unc}(S', M_S, O, O')$ having a prefix in $S' \cap (M_S - S)$ are irrelevant for the inequality above. Now consider a word w of $\text{Unc}(S', M_S, O, O')$ with no prefix in $S' \cap (M_S - S)$. Let w_p be the longest prefix of w which is in $S \cap S'$. There is a word $w' \in (O \cup O')^*$ and a symbol $\sigma \in O$ such that $w_p \circ w' \in S' \cap M_S$ and $w_p \circ w' \circ \sigma \in M_S - S'$. Suppose w' is not the empty string. Then we can let σ' be the first symbol of w'. Then $w_p \circ \sigma' \in M_S - S$, so $\sigma' \in O'$. But this means that $w \in \text{Unc}(S, S', O', O)$. If w' is empty, $w_p \in S \cap S'$ and $w_p \circ \sigma' \in M_S - S'$. Since $\sigma \in O$, $w_p \circ \sigma \in \cap M_S$ if and only if it belongs to S. Thus, $w_p \circ \sigma' \in S - S'$, which means that $w \in \text{Unc}(S', S, O, O')$. We can thus simplify the upper bound on E_R to

$$\begin{aligned}
U = \; &\begin{bmatrix} (S \cap S') \cup \\ \text{MissExt}(S, S', O') \cup \\ \text{MissExt}(S', S, O) \end{bmatrix} - \\
&\left[\text{Unc}(S', S, O, O') \cup \text{Unc}(S, S', O', O)\right].
\end{aligned} \tag{A.4}$$

Define $\hat{R} \stackrel{\text{def}}{=} (S \cap S') - [\text{Unc}(S', S, O, O') \cup \text{Unc}(S, S', O', O)]$. We want to show that $U = \hat{R} \cup \text{MissExt}(\hat{R}, \hat{R}, O \cup O')$. Note that we only have to prove that

$$\begin{aligned} \text{MissExt}(\hat{R}, \hat{R}, O \cup O') = \\ \begin{bmatrix} \text{MissExt}(S, S', O') \cup \\ \text{MissExt}(S', S, O) \end{bmatrix} - \\ [\text{Unc}(S', S, O, O') \cup \text{Unc}(S, S', O', O)]\,. \end{aligned} \tag{A.5}$$

Proof of (A.5). Suppose $w \in \text{MissExt}(S, S', O') - [\text{Unc}(S', S, O, O') \cup \text{Unc}(S, S', O', O)]$. Write $w = w_p \circ \sigma \circ w'$, where w_p is the longest prefix of w which lies in $S \cap S'$, $\sigma \in O'$, and $w' \in \Sigma^*$. $w_p \notin [\text{Unc}(S', S, O, O') \cup \text{Unc}(S, S', O', O)]$ because all its extensions would be in this set if w_p were in this set, and we know that w is not in this set. It follows that $w_p \in \hat{R}$ and since $w_p \circ \sigma \notin \hat{R}$, $w_p \circ \sigma$ and all its extensions are in $\hat{R} \cup \text{MissExt}(\hat{R}, \hat{R}, O \cup O')$. Thus, $w \in \hat{R} \cup \text{MissExt}(\hat{R}, \hat{R}, O \cup O')$

The same argument applies when

$$w \in \text{MissExt}(S', S, O) - [\text{Unc}(S', S, O, O') \cup \text{Unc}(S, S', O', O)]$$

. We conclude that the right hand side of (A.5) is a subset of the left hand side.

Now suppose that $w \in \text{MissExt}(\hat{R}, \hat{R}, O \cup O')$ and write $w = w_p \circ \sigma \circ w'$, where w_p is the longest prefix of w contained in $\hat{R}$, $\sigma \in O \cup O'$, and $w' \in \Sigma^*$. From the definition of $\hat{R}$, $w_p \in S \cap S'$. Suppose $w_p \circ \sigma \in S \cap S'$. Then

$$w_p \circ \sigma \in [\text{Unc}(S', S, O, O') \cup \text{Unc}(S, S', O', O)]\,,$$

which means that w_p also belongs to this set (because $\sigma \in O \cup O'$). This contradicts the fact that $w_p \in \hat{R}$, so our assumption that $w_p \circ \sigma \in S \cap S'$ is wrong. Then w_p is also the longest prefix of w contained in $S \cap S'$.

Without loss of generality, assume $\sigma \in O$. Suppose $w_p \circ \sigma \notin S$. We obtain $w \in \text{MissExt}(S', S, O)$. Moreover, since $w_p \in \hat{R}$, $w_p \circ \sigma \notin [\text{Unc}(S', S, O, O') \cup \text{Unc}(S, S', O', O)]$. Since $w_p \circ \sigma \notin S \cap S'$, we have $w \notin [\text{Unc}(S', S, O, O') \cup \text{Unc}(S, S', O', O)]$. Thus, w is in the right hand set of (A.5).

Now suppose $w_p \circ \sigma \notin S'$. If $w_p \circ \sigma \in S$, then $w_p \in \text{Unc}(S', S, O, O')$, which contradicts the fact that $w_p \in R$. We must have $w_p \circ \sigma \notin S$, which we already showed implies that w is in the right hand set of (A.5).

An analogous reasoning applies to $\sigma \in O'$. We conclude that the right hand side of (A.5) is a subset of the left hand side, and this finishes the proof of their equality. □

This result and (A.1) tell us that $E_R \leq \hat{R} \cup \text{MissExt}(\hat{R}, \hat{R}, O \cup O')$. Now we study the constraint (A.2). We want to show that $\hat{R}$ yields the tightest bound $L \leq \hat{R} \cup \text{MissExt}(\hat{R}, \hat{R}, I \cap I')$ which also respects the bound (A.1).

Proof. Observe that $L = (S' \cup \text{MissExt}(S', S', I')) \cap (S \cup \text{MissExt}(S, S, I))$. First we will show that $L \subseteq \hat{R} \cup \text{MissExt}(\hat{R}, \hat{R}, I \cap I')$. Suppose $w \in L$. Then w belongs to at least one of the sets (1) $S \cap S'$, (2) $S \cap \text{MissExt}(S', S', I')$, (3) $S' \cap \text{MissExt}(S, S, I)$, or (4) $\text{MissExt}(S, S, I) \cap \text{MissExt}(S', S', I')$. We analyze each case:

1. Suppose $w \in S \cap S'$. If $w \in \hat{R}$, then clearly $w \in \hat{R} \cup \text{MissExt}(\hat{R}, \hat{R}, I \cap I')$. Suppose $w \notin \hat{R}$. Then there is word $w' \in (O \cup O')^*$ and either a symbol $\sigma \in O$ such that $w \circ w' \circ \sigma \in S - S'$ or a symbol $\sigma \in O'$ such that $w \circ w' \circ \sigma \in S' - S$. Write $w = w_p \circ w''$ such that w'' is the longest suffix of w which belongs to $O \cup O'$. It follows that the last symbol of w_p is an element of $I \cap I'$. Since $w \notin \hat{R}$, neither does w_p. This shows that for every word $w_r \circ \sigma_r \in S \cap S'$ such that $w_r \in \hat{R}$ but $w_r \circ \sigma_r \notin \hat{R}$, we must have $\sigma_r \in I \cap I'$.

   Let w'_p be the longest prefix of w_p which lies in $\hat{R}$. By assumption, $\hat{R}$ is not empty. If we write $w = w'_p \circ w''$, the first symbol of w'' is in $I \cap I'$. Thus, $w \in \text{MissExt}(\hat{R}, \hat{R}, I \cap I')$.

2. Observe that

$$\text{MissExt}(S', S', I') = \text{MissExt}(S', S', I' \cap I) \cup \text{MissExt}(S', S', I' \cap O).$$

   Moreover,

$$S \cap \text{MissExt}(S', S', I \cap I') \subseteq \text{MissExt}(S \cap S', S \cap S', I \cap I') \tag{A.6}$$
$$\subseteq \text{MissExt}(\hat{R}, \hat{R}, I \cap I'). \tag{A.7}$$

   Suppose $w \in S \cap \text{MissExt}(S', S', I' \cap O)$. Then $w \in \text{Unc}(S', S, O, O')$, so $w \notin \hat{R}$. Let w_i be the longest prefix of w which lies in $S \cap S'$. Then $w_i \notin \hat{R}$, either. Let w_p be the longest prefix of w_i which is in $\hat{R}$. Then $w_i \in \text{MissExt}(\hat{R}, \hat{R}, I \cap I')$, and therefore, so does w.

3. If $w \in S' \cap \text{MissExt}(S, S, I)$, an analogous reasoning applies.

4. Suppose
$$w \in \text{MissExt}(S, S, I) \cap \text{MissExt}(S', S', I').$$
   If w has a prefix in $S' \cap \text{MissExt}(S, S, I)$ or $S \cap \text{MissExt}(S', S', I')$, then the reasoning of the last two points applies, and we have $w \in \text{MissExt}(\hat{R}, \hat{R}, I \cap I')$. Suppose w has no such a prefix, and write $w = w_p \circ w'$, where w_p is the longest prefix of w which lies in $S \cap S'$. Let σ be the first symbol of w'. Then $w_p \circ \sigma \in \text{MissExt}(S, S, I) \cap \text{MissExt}(S', S', I')$, which means that $\sigma \in I \cap I'$. Thus, $w \in \text{MissExt}(S \cap S', S \cap S', I \cap I') \subseteq \text{MissExt}(\hat{R}, \hat{R}, I \cap I')$.

We have shown that $L \subseteq \hat{R} \cup \text{MissExt}(\hat{R}, \hat{R}, I \cap I')$. Now suppose $w \in \hat{R} \cup \text{MissExt}(\hat{R}, \hat{R}, I \cap I')$. If $w \in \hat{R}$ then clearly $w \in S \cap S' \subseteq L$. Suppose $w \in \text{MissExt}(\hat{R}, \hat{R}, I \cap I')$ and let w_r be

the longest prefix of w contained in $\hat{R}$ and σ_r the symbol that comes immediately after w_r in w. Clearly $\sigma_r \in I \cap I'$.

If $w_r \circ \sigma_r \in S - S'$, then $w_r \circ \sigma_r$ cannot be an element of $\hat{R}$. If it were, we would have $E_{\hat{R}} \times S \not\subseteq E_{S'}$, violating the bound (A.1). The same applies when $w_r \circ \sigma_r \in S' - S$.

If $w_r \circ \sigma_r \notin S \cup S'$, then $w \circ \sigma_r \in \mathrm{MissExt}(S', S', I') \cap \mathrm{MissExt}(S, S, I) \subseteq L$.

If $w_r \circ \sigma_r \in S \cap S'$, then $w_r \circ \sigma_r \in \mathrm{Unc}(S', S, O, O') \cup \mathrm{Unc}(S, S', O', O)$, which means that $w_r \circ \sigma_r$ is not allowed to be an element of $\hat{R}$; otherwise, there would be a contradiction of (A.1). □

We conclude that $R = \hat{R}$. □

Proof of Proposition 8.5.8. Suppose that $A_1 \leq A_2$. We want to show that $M_{\ell(A_1)} \leq M_{\ell(A_2)}$ and $E_{\ell(A_2)} \leq E_{\ell(A_1)}$. We proceed by induction in the length n of words, i.e., we will show that this relations hold for words of arbitrary length.

Consider the case $n = 1$. Suppose $\sigma \in M_{\ell(A_1)} \cap \Sigma$. If $\sigma \in I$, then $\sigma \in M_{\ell(A_2)}$ because of I-receptivity. If $\sigma \in O$, then $\sigma \in \ell(A)$, so there exists $q_1 \in Q_1$ such that $q_{1,0} \xrightarrow{\sigma}_1 q_1$, which means that there exists $q_2 \in Q_2$ such that $q_{2,0} \xrightarrow{\sigma}_2 q_2$. Thus, $\sigma \in \ell(A_2) \subseteq M_{\ell(A_2)}$. We have shown that $M_{\ell(A_1)} \subseteq M_{\ell(A_2)}$ for $n = 1$. An analogous reasoning shows that $E_{\ell(A_2)} \subseteq E_{\ell(A_1)}$

Suppose the statement is true for words of length n. Let $w \circ \sigma \in M_{\ell(A_1)}$, where $w \in \Sigma^*$ is a word of length n, and $\sigma \in \Sigma$. By the inductive assumption, $w \in M_{\ell(A_2)}$.

- If $\sigma \in I$, then $w \circ \sigma \in M_{\ell(A_2)}$ due to I-receptiveness.

- Let $\sigma \in O$ and $w \notin \ell(A_1)$. Then we can write $w = w_p \circ w'$, where w_p is the longest prefix of w which lies in $\ell(A_1)$ (suppose it has length l). Let σ' be the first symbol of w'; clearly $\sigma' \in I$. Since $w \in \ell(A_2)$ and this set is prefix-closed, $w_p \in \ell(A_2)$. Since $w_p \in \ell(A_1) \cap \ell(A_2)$, there exist $\{q_{j,i} \in Q_j\}_{i=1}^{k}$ (for $j \in \{1, 2\}$) such that $q_{j,i-1} \xrightarrow{w_i}_j q_{j,i}$ for $0 < i \leq k$, where w_i is the i-th symbol of w_p. Since the IA are deterministic, we must have $q_{1,i} \leq q_{2,i}$. Suppose there were a $q_2 \in Q_2$ such that $q_{2,k} \xrightarrow{\sigma'}_2 q_2$; since $q_{1,k} \leq q_{2,k}$ and $\sigma' \in I$, this would mean that there exists $q_1 \in Q_1$ such that $q_{1,k} \xrightarrow{\sigma'}_1 q_1$, which would mean that $w_p \circ \sigma' \in \ell(A_1)$, a contradiction. We conclude that such q_2 does not exist, which means that $w_p \circ \sigma' \notin \ell(A_2)$, which means that $w \circ \sigma \in M_{\ell(A_2)}$ because of I-receptiveness.

- Finally, if $\sigma \in O$ and $w \in \ell(A_1)$, then there exist $\{q_{1,i} \in Q_1\}_{i=1}^{n}$ such that $q_{1,i-1} \xrightarrow{w_i}_1 q_{1,i}$ for $0 < i \leq n$, where w_i is the i-th symbol of w. Since $w \circ \sigma \in M_{\ell(A_1)}$, $w \in \ell(A_1)$, and $\sigma \in O$, we must have $w \circ \sigma \in \ell(A_1)$. This means that there must exist $q_{1,n+1} \in Q_1$ such that $q_{1,n} \xrightarrow{\sigma}_1 q_{1,n+1}$. We know that $w \in M_{\ell(A_2)}$ by the induction assumption. If $w \notin \ell(A_2)$, then clearly $w \circ \sigma \in M_{\ell(A_2)}$. If $w \in \ell(A_2)$, there are states $\{q_{2,i} \in Q_2\}_{i=1}^{n}$ such that $q_{2,i-1} \xrightarrow{w_i}_2 q_{2,i}$ for $0 < i \leq n$. Moreover, there exists $q_{n+1} \in Q_1$ such that $q_n \xrightarrow{\sigma}_1 q_{n+1}$ and $q_{1,n} \leq q_{2,n}$, there must be a $q_{2,n+1} \in Q_2$ such that $q_{2,n} \xrightarrow{\sigma}_2 q_{2,n+1}$, which means that $w \circ \sigma \in M_{\ell(A_2)}$.

We have shown that $M_{\ell(A_1)} \subseteq M_{\ell(A_2)}$. An analogous argument proves that $E_{\ell(A_2)} \subseteq E_{\ell(A_1)}$.

Now suppose that $M_{\ell(A_1)} \subseteq M_{\ell(A_2)}$ and $E_{\ell(A_2)} \subseteq E_{\ell(A_1)}$. We want to show that $q_{1,0} \leq q_{2,0}$. We proceed by coinduction.

Let n be a natural number. Suppose there exist sets $\{q_{j,i} \in Q_j\}_{i=1}^{n}$ with $j \in \{1,2\}$ such that $q_{1,i} \leq q_{2,i}$ for all i and a word w of length n such that $q_{j,i-1} \xrightarrow{w_i}_j q_{j,i}$ for $0 < i \leq n$. Suppose there exists $q_{1,n+1} \in Q_1$ and $\sigma \in O$ such that $q_{1,n} \xrightarrow{\sigma}_1 q_{1,n+1}$. Then $w \circ \sigma \in M_{\ell(A_1)} \subseteq M_{\ell(A_2)}$. Observe that $w \in \ell(A_2)$, so we must have $w \circ \sigma \in \ell(A_2)$. This means there must be a $q_{2,n+1} \in Q_2$ such that $q_{2,n} \xrightarrow{\sigma}_2 q_{2,n+1}$. We assume that $q_{1,n+1} \leq q_{2,n+1}$. Similarly, suppose there exists $q'_{2,n+1} \in Q_2$ and $\sigma \in I$ such that $q_{2,n} \xrightarrow{\sigma}_2 q'_{2,n+1}$. Then $w \circ \sigma \in E_{\ell(A_2)} \subseteq E_{\ell(A_1)}$. Since $w \in \ell(A_1)$, we must have $w \circ \sigma \in \ell(A_1)$. Thus, there must exist $q'_{1,n+1} \in Q_1$ such that $q_{1,n} \xrightarrow{\sigma}_1 q'_{1,n+1}$. We assume that $q_{1,n+1} \leq q_{2,n+1}$. This finished the coinductive proof. □

Proof of Proposition 8.5.9. Let A_i have IO signatures (I_i, O_i) for $i \in \{1,2\}$. For composition to be defined, we need $I_1 \cup I_2 = \Sigma$. Let $\mathcal{C}_{A_i}$ be the interface contract associated with A_i. From Proposition 8.5.7 and Section 8.5.3, the composition $\mathcal{C}_{A_1} \parallel \mathcal{C}_{A_2}$ is isomorphic to $I_1 \cap I_2$ and the $\mathcal{L}_\emptyset$ language

$$R = (\ell(A_1) \cap \ell(A_2)) - [\mathrm{Unc}(\ell(A_1), \ell(A_2), O_2, O_1) \cup \mathrm{Unc}(\ell(A_2), \ell(A_1), O_1, O_2)].$$

From Section 8.5.5.2, we deduce that $\ell(A_1 \parallel A_2) = R$. The proposition follows. □