Contracts for System Design

Albert Benveniste  
INRIA, Rennes, France  
albert.benveniste@inria.fr

Benoît Caillaud  
INRIA, Rennes, France  
benoit.caillaud@inria.fr

Dejan Nickovic  
Austrian Institute of Technology  
Dejan.Nickovic@ait.ac.at

Roberto Passerone  
University of Trento, Italy  
roberto.passerone@unitn.it

Jean-Baptiste Raclet  
IRIT, Toulouse, France  
raclet@irit.fr

Philipp Reinkemeier  
Offis, Oldenburg, Germany  
Philipp.Reinkemeier@offis.de

Alberto Sangiovanni-Vincentelli  
University of California at Berkeley  
alberto@eecs.berkeley.edu

Werner Damm  
Offis and University of Oldenburg, Germany  
werner.damm@offis.de

Thomas A. Henzinger  
IST Austria, Klosterneuburg  
tah@ist.ac.at

Kim G. Larsen  
Aalborg University, Denmark  
kgl@cs.aau.dk
Contents

1 Introduction 125
  1.1 Industrial context ........................................... 125
  1.2 Positive impact of contract-based design ...................... 127
  1.3 A bird’s eye view of research in contracts .................. 130
  1.4 Contribution of this monograph ............................ 131

  2.1 Contract based design ....................................... 137
  2.2 A primer on contracts ........................................ 141

3 Positioning of this Monograph and Bibliographical Note 151
  3.1 Contracts in software engineering ............................ 151
  3.2 Contracts for (possibly cyber-physical) systems ............. 157

4 A Mathematical Meta-Theory of Contracts 160
  4.1 Components .................................................. 161
  4.2 Contracts ................................................... 163
  4.3 Refinement and conjunction .................................. 164
  4.4 Parallel composition ........................................ 166
  4.5 Quotient ..................................................... 172
  4.6 Making contract composition associative .................... 172
  4.7 Abstractions ................................................ 173
4.8 Bibliographical note on abstract contract theories .................................. 179

5 Assume/Guarantee Contracts ........................................................................ 182
  5.1 Synchronous A/G contracts with fixed alphabet ...................................... 183
  5.2 Dealing with variable alphabets .............................................................. 189
  5.3 Abstractions ............................................................................................. 190
  5.4 Observers .................................................................................................. 191
  5.5 Asynchronous dataflow A/G contracts .................................................... 195
  5.6 A/G contracts for Cyber-Physical systems .............................................. 196
  5.7 Discussion ................................................................................................. 197
  5.8 Bibliographical note .................................................................................. 198

6 Synchronous Moore Interfaces and A/G Contracts .................................... 204
  6.1 Introduction ............................................................................................... 204
  6.2 An illustration example for Moore Interfaces ......................................... 205
  6.3 A/G contract saturation via Moore Interfaces ......................................... 208
  6.4 Moore Interfaces, seen as A/G contracts ................................................ 212
  6.5 Discussion ................................................................................................. 215

7 Rely/Guarantee Reasoning and A/G Contracts ............................................ 217
  7.1 A brief on rely/guarantee reasoning .......................................................... 217
  7.2 Components for shared variable concurrency ........................................ 219
  7.3 Contracts for shared variable concurrency .............................................. 224
  7.4 Discussion ................................................................................................. 227

8 Interface Theories ......................................................................................... 228
  8.1 Components as i/o-automata .................................................................... 229
  8.2 Interface Automata with fixed alphabet .................................................. 231
  8.3 Modal Interfaces with fixed alphabet ........................................................ 242
  8.4 The approach by Gerald Lüttgen, Walter Vogler et al. ............................ 268
  8.5 Modal Interfaces with variable alphabet .................................................. 270
  8.6 Decomposing a contract as a composition of subcontracts ...................... 272
  8.7 Modal interfaces as Assume/Guarantee contracts .................................... 276
  8.8 Bibliographical note .................................................................................. 283

9 Scheduling Contracts ...................................................................................... 292
  9.1 Introduction ............................................................................................... 292
9.2 Scheduling components ............................................. 298
9.3 Scheduling contracts ............................................... 312
9.4 Sub-contracting in the development process .................... 316
9.5 Modeling methodology ............................................. 321
9.6 Bibliographical note .................................................. 324

10 Contracts for Requirement Engineering .............................. 328
   10.1 Motivation: formalizing requirements ......................... 328
   10.2 The car parking system, informal presentation ............... 331
   10.3 Formalization using contracts ................................. 335
   10.4 Discussion ..................................................... 347

11 Contracts for Timing in Autosar ...................................... 350
   11.1 Motivation: timing issues in Autosar ......................... 350
   11.2 An example of an Autosar design process using scheduling contracts ............................................. 351
   11.3 Summary and discussion ....................................... 360
   11.4 Bibliographical note ............................................ 361

12 Conclusion .................................................................. 363
   12.1 Status of research ................................................. 363
   12.2 Status of practice ................................................ 364
   12.3 Advances in contract theories .................................. 368
   12.4 Application of contracts: lessons from our experiments .... 370
   12.5 Epilogue ........................................................ 372

Acknowledgements ......................................................... 374

References ................................................................... 375
Abstract

Recently, contract-based design has been proposed as an “orthogonal” approach that complements system design methodologies proposed so far to cope with the complexity of system design. Contract-based design provides a rigorous scaffolding for verification, analysis, abstraction/refinement, and even synthesis. A number of results have been obtained in this domain but a unified treatment of the topic that can help put contract-based design in perspective was missing. This monograph intends to provide such a treatment where contracts are precisely defined and characterized so that they can be used in design methodologies with no ambiguity. In particular, this monograph identifies the essence of complex system design using contracts through a mathematical “meta-theory”, where all the properties of the methodology are derived from a very abstract and generic notion of contract. We show that the meta-theory provides deep and illuminating links with existing contract and interface theories, as well as guidelines for designing new theories. Our study encompasses contracts for both software and systems, with emphasis on the latter. We illustrate the use of contracts with two examples: requirement engineering for a parking garage management, and the development of contracts for timing and scheduling in the context of the AUTOSAR methodology in use in the automotive sector.
1

Introduction

1.1 Industrial context

System companies such as automotive, avionics and consumer electronics enterprises are facing significant difficulties due to the exponentially raising complexity of their products coupled with increasingly tight demands on functionality, correctness, and time-to-market. The cost of being late to market or of imperfections in the products is staggering as witnessed by the recent recalls and delivery delays that system industries had to bear. Many challenges face the system community to deliver products that are reliable and effective. Table 1.1, albeit not recent, continues to be a telling example of the main causes and their share in the difficulties related to systems complexity.¹ This table highlights the importance of system integration, where corrections occur late in the design flow and are therefore very costly.

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

Table 1.1: Difficulties related to system complexity. The table displays, for each industrial sector, the percentage of tasks delayed and tasks causing delays, for the different phases of system design.

<table>
<thead>
<tr>
<th>Design task</th>
<th>Tasks delayed automotive</th>
<th>Tasks delayed automation</th>
<th>Tasks delayed medical</th>
</tr>
</thead>
<tbody>
<tr>
<td>System integration test, and verification</td>
<td>63.0%</td>
<td>56.5%</td>
<td>66.7%</td>
</tr>
<tr>
<td>System architecture design and specification</td>
<td>29.6%</td>
<td>26.1%</td>
<td>33.3%</td>
</tr>
<tr>
<td>Software application and/or middleware development and test</td>
<td>44.4%</td>
<td>30.4%</td>
<td>75.0%</td>
</tr>
<tr>
<td>Project management and planning</td>
<td>37.0%</td>
<td>28.3%</td>
<td>16.7%</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Design task</th>
<th>Tasks causing delay automotive</th>
<th>Tasks causing delay automation</th>
<th>Tasks causing delay medical</th>
</tr>
</thead>
<tbody>
<tr>
<td>System integration test, and verification</td>
<td>42.3%</td>
<td>19.0%</td>
<td>37.5%</td>
</tr>
<tr>
<td>System architecture design and specification</td>
<td>38.5%</td>
<td>42.9%</td>
<td>31.3%</td>
</tr>
<tr>
<td>Software application and/or middleware development and test</td>
<td>26.9%</td>
<td>31.0%</td>
<td>25.0%</td>
</tr>
<tr>
<td>Project management and planning</td>
<td>53.8%</td>
<td>38.1%</td>
<td>37.5%</td>
</tr>
</tbody>
</table>


platforms. In addition, 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. Examples of aspects are system architecture, the functions the system should perform and the services it should deliver, its safety and reliability characteristics, its energy budget, its deployment on an embedded computing platform to name a few.

*Contract-based design* has as main goal solving the above issues in a rigorous framework.

### 1.2 Positive impact of contract-based design

**Addressing the Complexity of Systems.** Several approaches have been developed by research institutions and industry to cope with the exponential growth in systems complexity. Of particular interest to the development of embedded controllers and systems are *layered design* and *component-based design* (used, e.g., in the AUTOSAR\(^2\) standard in the automotive sector, and the ARINC\(^3\) standard in the avionic domain), *model-based development* (supported by important frameworks and tools such as SysML\(^4\) [208] and/or AADL [211] for architecture modeling, and Modelica [133] and Matlab-Simulink [168] for system modeling), *virtual integration* (Ptolemy [124] and Metropolis [100, 68]), and *platform-based design* [100, 114, 221]. There are two basic principles followed by these methods: abstraction/refinement and composition/decomposition. Abstraction and refinement are processes that relate to the flow of design between different layers of abstraction (vertical process) while composition and decomposition operate at the same level of abstraction (horizontal process). Layered design and model-based development focus on the vertical process while component-based design deals principally with the horizontal process. Platform-based design combines the two aspects in a unified framework.

While the above methods have been critical steps in breaking systems complexity, they do not by themselves provide the ultimate answer. Contracts are ideal tools to solidify both vertical and horizontal processes providing the theoretical background to support formal methods in system design. When design is being performed at a considered layer, implicit — and often hidden — assumptions regarding other layers (e.g., computing resources)

\(^2\)http://www.autosar.org/
\(^3\)https://www.aviation-ia.com/product-categories/arinc
\(^4\)http://www.omg.org/spec/SysML/
are typically invoked by the designer. Actual properties of these other layers, however, cannot be compared against these hidden assumptions. Similarly, when components or sub-systems are abstracted via their interfaces in component based design, it is generally not true that such interfaces provide sufficient information for other components to be safely implemented based on this sole interface. By pinpointing responsibilities and making hidden assumptions explicit, contract-based design provides the due discipline, concepts, and techniques to cope with this.

Another challenge for component-based design of embedded systems is to provide interface specifications that address behaviors, not only type properties of interfaces, and are rich enough to cover all phases of the design cycle. This calls for including non-functional characteristics as part of the component interface specifications, which is best achieved by using multiple viewpoints [40, 46, 42]. Contract-based design supports multiple viewpoints by giving a mathematically precise answer to what it means to fuse them.

### Addressing OEM-Supplier Chains and Managing Requirements.

The management of responsibilities in and design processes across OEM-supplier chains is indeed the core target of contract-based design. By making the explication of implicit assumptions mandatory, contracts help assign responsibilities to a precise stake holder for each design entity. By supporting independent development of the different sub-systems while guaranteeing smooth system integration, they orthogonalize the development of complex systems. Contracts are thus adequate candidates for a technical counterpart of the legal bindings between partners involved in the distributed and concurrent development of a system.

Regarding requirement capture, efforts have been made by paying close attention to book-keeping activities such as the management of the requirement descriptions and corresponding traceability support (e.g., using commercial tools such as Doors\(^5\) in combination with Reqtify\(^6\)) and by inserting, whenever possible, precise formulation and analysis methods and tools. Still, the need for basing requirement engineering on more solid bases is widely

---

\(^6\)https://www.3ds.com/fr/produits-et-services/catia/produits/reqtify/
1.2. Positive impact of contract-based design

acknowledged. Specifications used for procurement should be precise, unambiguous, and complete. Indeed, a recurrent reason for failures causing deep iterations across supply chain boundaries rests in incomplete characterizations of the conditions for use and environment of the system to be developed by the supplier, such as missing information about failure modes and failure rates, missing information on possible sources of interference through shared resources, and missing boundary conditions. This argument highlights the need of making assumptions on the design context explicit in OEM-supplier commercial contracts. The potentially highest value proposition of a systematic introduction of contracts indeed lies in requirement capture. Already the evaluation results of the industrial partners in the Integrated Project Speeds\(^7\) acclaim the use of contracts for the requirement capture phase to substantially increase the quality of requirements.

By systematically enforcing the explication of assumptions, systems understanding and thus system interface specifications are substantially improved. Thinking in terms of assumptions uncovers early potential incompatibilities, which otherwise would have only been found much later in integration stages. Furthermore, (i) the explication of assumptions significantly eases concurrent engineering; assumptions provide a natural way of communication between design teams; (ii) the quality improvements in requirements translates directly to improvement of test cases for requirement-based testing; and (iii) the effort spent in explicating assumptions translates directly to improvement of test cases for integration testing. Assumptions are easily integrated into industrial design flows for requirement capture, including tools for traceability and change management. Further, formalized contracts allow for a rigorous checking of otherwise easily overlooked inconsistencies between requirements. Formalized contracts allow for “playing out” contracts — a term coined by David Harel [149, 147] — i.e., executing formalized specifications by engines that systematically generate all behaviours possible under the current set of contracts. Such simulation based environments give strong support for checking the completeness of requirements. Finally, vectors for requirement based testing and virtual integration testing can be automatically derived from formalized contracts, again leading to a significant quality improvement. Observers can be automatically generated from

\(^7\)http://cordis.europa.eu/project/rcn/79466_en.htm
formalized contracts and used in model-, software- and hardware-in-the-loop testing, or even integrated into execution platforms e.g. to diagnose failure situations. We will thus in this monograph elaborate in particular on the benefits of formalized contracts for requirement capture.

1.3 A bird’s eye view of research in contracts

The notion of contract is not new. It was first developed and promoted in the community of software engineering, and more specifically Model Driven Engineering. Actually, Design by Contract is a software engineering technique popularized by Bertrand Meyer [200, 201] following earlier ideas from Floyd-Hoare logic [234, 155]. Floyd-Hoare logic assigns meaning to sequential imperative programs in the form of triples of assertions consisting of a precondition on program states and inputs, a command, and a postcondition on program states and outputs. So far contracts consisting of pre/postconditions naturally fit imperative sequential programming. In situations where programs may operate concurrently, interference on shared variables can occur. Rely/Guarantee rules [159] were thus added to interface contracts. Rely conditions state assumptions about any interference on shared variables during the execution of operations by the system’s environment. Guarantee conditions state obligations of the operation regarding shared variables.

Despite early contributions by Abadi, Lamport, and Wolper [5, 3], developing contracts for Cyber-Physical Systems [236, 100] and Reactive Systems [146, 152, 142, 196], where mathematical behaviors are essential, boomed more recently in the 2000’s, when de Alfaro and Henzinger proposed and popularized so-called interface theories [105, 103, 8]. Since then, a number of models have been proposed that can be seen as instances of contract theories, either to address a specific technical aspect (e.g., function, timing, and resources), or by following different styles and approaches (Assume/Guarantee contracts or Interfaces).9

8Distributed physical systems complemented by computing systems.
9See the dedicated bibliographical notes in this monograph.
1.4 Contribution of this monograph

This wide diversity in the proposed approaches calls for a clarification of what the essence of a contract theory is. More specifically, we need an abstract and generic theory (a *meta-theory*) of contracts or interfaces that abstracts away how contracts and actual designs are actually represented and still formally defines the following concepts:

- *Implementations and environments* that conform to the contract; a contract is *consistent* if it possesses legal implementations and *compatible* if it possesses legal environments;

- contract *refinement*, the proper notion of substitutability for contracts;

- *Conjunction* of contracts, how to “fuse” different viewpoints;

- *Parallel composition* of contracts, how composing (sub-)contracts attached to subsystems yields a system-level contract; the aim is that this parallel composition supports independent development, meaning in particular that composing legal implementations for each subcontract yields a legal implementation for the system-level contract;

- an additional, less essential but still useful concept, is that of *quotient*, which is the adjoint of the notion of parallel composition; how to “patch” an existing design to make it satisfy a new contract.

As the central contribution of this monograph, we thus propose a mathematical *meta-theory of contracts* and specialize it to different existing contract theories and variations thereof. In addition to presenting a number of new results, the monograph has a tutorial value in explaining the role of contracts and interfaces in design. In this respect, we include extensive bibliographical notes with particular attention to the numerous results published since year 2000. Since a number of topics are addressed, we preferred to defer bibliographical studies to the different chapters for each different topic.

The monograph is organized as shown in the Figure 1.1, which shows a dependency map between the different chapters. Chapters 1, 2, and the concluding Chapter 12 address readers who may not be specialists in contracts nor on formal methods (except for the summary of results when describing
the organization of the paper). Chapter 3 is a wide scope discussion of the state of the art—details of recent results are not discussed. Chapter 4, which is a key contribution of this monograph, is more technical but is meant to be self-contained and should be readable by anyone having general skills in mathematics. Chapters 5 to 9 target readers enough exercised in formal methods and, for some parts, even researchers in the field. Some readers may be particularly interested in a particular contract framework and then concentrate on the corresponding chapter. Alternatively, she may be interested in links between frameworks. The two application Chapters 10 and 11 target a wider audience, although they rely on the technical material of previous chapters. A more detailed description of these chapters follows.

![Diagram of chapter dependencies](image-url)

**Figure 1.1:** Organization of the monograph and dependencies between chapters.
1.4. Contribution of this monograph

In Chapter 2 we first discuss the requirements on a theory of contracts, based on methodological considerations. In particular we stress the need to support different viewpoints on the system (e.g., operation, function, timing, energy, safety). Then we develop a primer on contracts by using a very simple example requiring only elementary mathematical background. The purpose of this simplistic example is to smoothly and informally introduce the different concepts and operations we need for a contract framework.

Chapter 3 presents a birds eye bibliography of the subject and explains the positioning of our work. So far the links and parallels between the two notions of contract in Object Oriented programming and contract or interface for system design were obscure. In this chapter we draw these two landscapes and pave the way for clarifying the (actually existing) links between these two notions of contract.

Chapter 4 is the cornerstone of this monograph: It presents a new vista on contracts. The so-called “meta-theory” of contracts is introduced and developed in detail. By meta-theory we mean the collection of concepts, operations, and properties that any formal contract framework should offer. Every concrete framework compliant with this meta-theory will inherit these generic properties. The principle of the meta-theory is the definition of a contract as two sets: correct implementations and legal environments. In doing so, we do not assume any particular way of specifying implementations or environments thus making the meta-theory applicable to any contract theory proposed in the literature. Architecture design is greatly facilitated if the framework used allows to re-structure in a different way a system architecture, while preserving its overall semantics (i.e., meaning). A mathematical formalization of this feature is by requiring that the composition operator supporting architecture modeling shall be associative: \((M \times M') \times M'' = M \times (M' \times M'')\), illustrated in Figure 1.2. When applied to contracts, the same property is key in supporting independent development of subsystems by different suppliers with safe system integration. The meta-theory naturally leads, instead, to the consideration of a weaker notion of sub-associativity, involving the refinement for its definition — we prove that sub-associativity is sufficient for supporting independent development. Not all concrete contract frameworks possess an associative parallel composition; our results prove that sub-associativity nevertheless holds. We give a tight additional axiom for the
Introduction

Figure 1.2: Illustrating associativity in architectures. The figures show two architectures using the same set of five components. Components are grouped into different subsystems in the top and bottom architectures. Associativity means that the two obtained architectures should possess identical semantics.

meta-theory ensuring that contract composition is associative and we show that this axiom holds for Assume/Guarantee contracts. We introduce the notion of quotient, which supports the practice of patching an existing system to make it satisfy different specifications; the quotient formalizes the concept of “minimal patch”. We finally show how abstraction techniques can be defined at the level of the meta-theory, thus specializing to any compliant contract theory. The meta-theory does not specify how components and contracts are effectively represented and manipulated. The subsequent series of chapters presents a panorama of major concrete contract frameworks.

Chapter 5 deals with Assume/Guarantee contracts [40, 46]. This framework is the most straightforward instance of the meta-theory. It presents pairs \((A, G)\) of assumptions and guarantees explicitly, \(A\) and \(G\) being both expressed as properties. This framework is flexible in that it allows for different styles of description of such properties — computational efficiency depends on the style adopted. In Chapter 6 we relate the Synchronous Interfaces [82]
1.4. Contribution of this monograph

to the Assume/Guarantee contracts. In Chapter 7 we analyse Rely/Guarantee reasoning [159] used in the area of software engineering and formal methods, to reason about concurrency. We show that this reasoning is also tightly related to Assume/Guarantee contracts. Chapter 8 develops the Interface theories [105, 19], in which assumptions and guarantees are specified by means of a single object: the interface. We revisit the notion of quotient for Modal Interfaces [226, 227], to make it the proper specialization of the notion of quotient following the meta-theory. We use this revisited quotient in a holistic methodology for automatically moving from system-level requirements to a set of subcontracts for the different suppliers. We ground on firm bases how Assume/Guarantee contracts can be emulated using Modal Interfaces. Chapter 9 develops a contract framework addressing schedulability analysis, a task involving resource aspects. This framework is subtle because the time and the computing resources both have a strong global flavor.

We complement the above chapters devoted to aspects of the theory with two illustration cases. In Chapter 10, we develop and study requirements for a simple parking garage. Its top-level specification comprises several viewpoints, each one consisting of a requirement table. We pay attention to responsibilities by properly identifying assumptions regarding the environment (context of use), and guarantees offered by the system if properly used. We then study the critical design step consisting in producing sub-contracts for each supplier, following an architecture of sub-systems that differs from the top-level architecture — a frequently encountered situation. We go beyond the state-of-the-art by proposing a synthesis method and algorithm, by which the sub-contracts are automatically derived, from the top-level contract and the (SysML-like) topological description of the sub-systems architecture. We discuss the use of contracts in formally establishing properties of the requirements such as consistency, compatibility, and completeness. Despite this being a simple example, it is yet much too complex to be dealt with by hand. A Proof of Concept tool was used to support our development. The contract framework used for this study is the Modal Interfaces.

Chapter 11, which is intended to present an industrially-relevant application, addresses a key part of the AUTOSAR development process in use in the automotive industry. AUTOSAR advocates a design methodology by which the functions, structured into tasks, are first designed independently of the
computing and communication infrastructure, assuming a virtual AUTOSAR run time environment. We study the key step by which time budgets are then allocated to tasks and computing resources are assigned. Lack of formal support in AUTOSAR methodology makes this step difficult today. We show the benefit of using contracts for this step. To this end, we develop an adaptation, called scheduling contracts, of the Assume/Guarantee contracts.

Finally the concluding chapter summarizes the lessons drawn from this work and analyzes the industrial situation.
There are two basic principles followed by design methods so far developed, namely: abstraction/refinement and composition/decomposition. Abstraction and refinement are processes that relate to the flow of design between different layers of abstraction (vertical process) while composition and decomposition operate at the same level of abstraction (horizontal process). In this section we motivate by methodological considerations the kind of property we expect from contracts. We then study a simple instance of such a contract framework on a toy example, where all operations can be exemplified. We conclude this chapter by providing a (non exhaustive) bibliography on the general concept of contract.

2.1 Contract based design

Contract based design can be seen as a set of methodological guidelines exploiting a framework of contracts characterized by the operations of refinement, conjunction, and composition, denoted in the sequel by the symbols $\preceq$, $\land$, and $\otimes$, respectively. In this section we review these guidelines and discuss the requirements they set about the contract framework.
**Contracts: What? Where? And How?**

Supporting open systems: Component reuse requires that components be seen as open entities, meaning that their context of use is not fully known while the component is being designed. We therefore need a description of components in which both the guarantees offered by the component and the assumptions on its possible context of use, we call it its environment, shall be exposed. This states what contracts should be.

Managing Requirements and Fusing Viewpoints: Requirements aim at specifying properties that are expected from the system under design. In particular, requirements are the mean by which an OEM interacts with its suppliers, on both a legal and a technical perspective. Properties expected from the system are called guarantees. Each guarantee \( G \) relies on a set \( \{A_1, \ldots, A_\ell\} \) of properties called assumptions, expressing boundary conditions or legal contexts of use for the system in order for the guarantee \( G \) to hold.

Such assumptions are often left implicit, which is both a source of problems at system integration, and a source of dispute regarding liability between the OEM and its suppliers in case a problem occurs.

How should assumptions combine, versus how should guarantees combine, in a requirements document? Clearly, guarantees must combine in a conjunctive way. This is reflected by the common practice that “the system must pass all tests attached to the different requirements”. What about assumptions? If the system is used in a way that violates some assumption, then the system is relieved from the set of guarantees that relied on this assumption. Other guarantees, however, remain. In fact, each guarantee \( G \) should be considered along with its associated set \( \{A_1, \ldots, A_\ell\} \) of assumptions and interpreted as the entailment

\[
\{A_1, \ldots, A_\ell\} \Rightarrow G. \tag{2.1}
\]

The right interpretation of a requirement document is thus the conjunction of all such entailments — clearly, assumptions must not be conjuncted. Therefore, contract theories need to offer a notion of conjunction that interprets requirements documents in the above way.

This very same concept should also be valid to set the meaning of how the combination of different chapters or viewpoints of the systems requirements
must be interpreted. Typical instances of viewpoints are function, safety, energy, etc. These viewpoints rely on different modeling frameworks but nevertheless generally interact. The use of contracts for managing requirements is illustrated in Figure 2.1.

![Figure 2.1: Conjunction of requirements and viewpoints in top-level design](image)

**Design Chain Management, Reuse, and Independent Development:**
In Figure 2.2, we show three successive stages of the design. At the top level sits the overall system specification as developed by the OEM. This has the form of a contract $C$. As an example, $C$ can be obtained as the conjunction of several viewpoints as illustrated on Figure 2.1.

As a first design step, the OEM decomposes its system into an architecture made of three subsystems for independent development by (possibly different) suppliers. For each of these subsystems, a contract $C_j$, $j = 1, 2, 3$ is developed. A contract composition, denoted by the symbol “⊗”,

$$C_1 \otimes C_2 \otimes C_3$$

mirrors the composition of subsystems that defines the architecture. For our method to support independent development, this contract composition operator must satisfy the following:

*if designs are independently performed for each subcontract $C_j$, for $j = 1, 2, 3$, then integrating the delivered subsystems yields an implementation that satisfies the composed contract $C_1 \otimes C_2 \otimes C_3$. (2.2)*

This contract composition must then be qualified against the top-level contract $C$. This qualification must ensure that any development compliant with

\[ C_1 \otimes C_2 \otimes C_3 \] should also comply with \( C \). To ensure substitutability, compliance concerns both how the system behaves and what its valid contexts of use are: any valid context for \( C \) should be valid for \( C_1 \otimes C_2 \otimes C_3 \) and, under such context, the integrated system should behave as specified by \( C \). This will be formalized as the refinement relation, denoted by the symbol \( \preceq \):

\[ C_1 \otimes C_2 \otimes C_3 \preceq C \quad (2.3) \]

Overall, the satisfaction of (2.2) and (2.3) guarantees the correctness of this first design step performed by the OEM.
2.2 A primer on contracts

Obtaining the three subcontracts $C_1$, $C_2$, and $C_3$, is the art of the designer, based on architectural considerations. Contract theories, however, offer the following services to the designer:

- The formalization of parallel composition and refinement for contracts allows the designer to firmly assess whether (2.3) holds for the decomposition step or not.
- The compatibility of the three subcontracts $C_1$, $C_2$, and $C_3$, can be formally checked.
- Using contracts as a mean to communicate specifications to suppliers guarantees that the information provided to the supplier is self-contained. The supplier has all the information it needs to develop its subsystem so that subsequent system integration will be correct.

Each supplier can then proceed with the independent development of the subsystem it is responsible for. For instance, a supplier may reproduce the above described “top-down” procedure.

Alternatively, this supplier can develop some subsystems by reusing off-the-shelf components. For example, contract $C_{22}$ would be checked against the interface specification of a pre-defined component $M_{22}$ available from a library, and the following would have to be verified: does component $M_{22}$ satisfy $C_{22}$? In this context, shared implementations are of interest. This is illustrated on Figure 2.3 where, by implementing the contract $C_{22} \land C_{32}$ a same off-the-shelf component $M_{2/3,2}$ (not shown) implements $C_{22} \land C_{32}$, hence it implements the two contracts $C_{22}$ and $C_{32}$.

To conclude on this analysis, the two notions of refinement, denoted by the symbol “$\leq$”, and composition of contracts, denoted by the symbol “$\otimes$”, are key. Condition (2.2) ensures that independent development holds.

2.2 A primer on contracts

In this section we instantiate the above motivated calculus on a very simple framework of “stateless contracts” where the properties considered do not involve system states. This section is purposely intuitive and informal. Formal developments begin with Chapter 4.
2.2.1 Components, Environments, and Contracts

We start from a model that consists of an underlying set $M$ of components, each denoted by the symbol $M$. A component $M$ is typically an open system, i.e., it contains some inputs that are provided by other components in the system or the external world and it generates some outputs. This collection of other components and the exterior world is referred to as the environment of the component. The environment is often not completely known when the component is being developed. Although components cannot constrain their environment, they are designed to be used in a particular context.

In the following example, we define a component $M_1$ that computes the division between two real inputs $x$ and $y$, and returns the result through the real output $z$. The underlying assumption is that $M_1$ will be used within a design context that prevents the environment from giving the input $y = 0$. Since $M_1$ cannot constrain its input variables, we handle the unwanted input $y = 0$ by generating an arbitrary output (0 in this case):

$$ M_1 : \begin{cases} \text{variables:} & \begin{cases} \text{inputs:} & x, y \\ \text{outputs:} & z \end{cases} \\ \text{types:} & x, y, z \in \mathbb{R} \\ \text{behaviors:} & (y \neq 0 \rightarrow z = x/y) \land (y = 0 \rightarrow z = 0) \end{cases} $$
A contract, denoted by the symbol $\mathcal{C}$, is a way of specifying components with the following characteristic properties:

1. Contracts are intentionally abstract;

2. Contracts distinguish responsibilities of a component from those of its environment.

By property 1, contracts expose enough information about the component, but not more than necessary for the intended purpose. We can see a contract as an under-specified description of a component that can either be very close to the actual component, or specify only a single property of a component behavior. Regarding property 2, and in contrast to components, a contract explicitly makes a distinction between assumptions made about the environment, and guarantees provided, mirroring different roles and responsibilities in the design of systems.

A contract can be implemented in a number of different ways and can operate in a number of different environments. Hence, at its most abstract level, we define a contract $\mathcal{C}$ as a pair $\mathcal{C} = (\mathcal{E}_C, \mathcal{M}_C)$ of subsets of implementations of it and of subsets of environments in which it can operate. We say that a contract $\mathcal{C}$ is consistent if it possesses implementations, i.e., $\mathcal{M}_C \neq \emptyset$, and compatible if there exists environments in which it can operate, i.e., $\mathcal{E}_C \neq \emptyset$. Sets $\mathcal{E}_C$ and $\mathcal{M}_C$ are generally infinite. In concrete contract-based design theories, however, a contract needs to have a finite description that does not consist of its actual sets of legal implementations and environments. Moreover, the implementation relation needs to be effectively computable and establish the desired link between a contract and the components that implement it or are legal environments for it.

For our present simple example of static systems, we propose the following way to specify contracts:

\[
\mathcal{C}_1 : \begin{cases} 
\text{variables:} & \{ \text{inputs: } x, y, \text{ outputs: } z \} \\
\text{types:} & x, y, z \in \mathbb{R} \\
\text{assumption } A_1 : & y \neq 0 \\
\text{guarantee } G_1 : & z = x/y
\end{cases}
\]
The assumption is specified as the property \( A_1 : y \neq 0 \) and the guarantee is specified as the property \( G_1 : z = x/y \). \( \mathcal{C}_1 \) defines the set of components having \( x, y \) as input variables and \( z \) as output variable, all of type real, and whose behaviors satisfy the implication
\[
A_1 \Rightarrow G_1
\]
i.e., for the above example, \( y \neq 0 \Rightarrow z = x/y \). Intuitively, contract \( \mathcal{C}_1 \) specifies the intended behavior of components that implement division. It explicitly makes the assumption that the environment will never provide the input \( y = 0 \) and leaves the behavior for that input undefined.

This contract describes an infinite number of environments in which it can operate, namely the set \( \mathcal{E}_{\mathcal{C}_1} \) of environments providing values for \( x \) and \( y \), with the condition that \( y \neq 0 \). It describes an infinite number of components that implement the above specification, where the infinity comes from the underspecified case on how an implementation of \( \mathcal{C}_1 \) should cope with the illegal input \( y = 0 \). In particular, we have that \( M_1 \) implements \( \mathcal{C}_1 \). Thus, contract \( \mathcal{C}_1 \) is consistent. We now show a variant of contract \( \mathcal{C}_1 \) that is not consistent:

\[
\mathcal{C}_1' : \left\{ \begin{array}{l}
\text{variables:} \quad \{ \text{inputs: } x, y \\
\quad \text{outputs: } z \\
\text{types:} \quad x, y, z \in \mathbb{R} \\
\text{assumption } A_1' : \quad \top \\
\text{guarantee } G_1' : \quad z = x/y
\end{array} \right.
\]

where symbol \( \top \) denotes the boolean constant “true”. In contrast to \( \mathcal{C}_1 \), the contract \( \mathcal{C}_1' \) makes no assumption on values of the input \( y \). Hence, every component that implements \( \mathcal{C}_1' \) has to compute the quotient \( x/y \) for all values of \( y \), including \( y = 0 \), which makes no sense.

### 2.2.2 Contract Operators

There are three basic contract operators that are used in support of the design methodologies we presented previously: **composition**, **refinement** and **conjunction**.

**Contract Composition and System Integration:** Intuitively, the composition operator acting on components supports component-based design.
2.2. A primer on contracts

This composition operator, which we denote by the symbol $\times$, is a partial function on components. The composition is defined with respect to a composability criterion: for our illustration example, two components $M$ and $M'$ are *composable* if their variable types match and if they do not share output variables. Generally, composability is a syntactic property on pairs of components that defines conditions under which the two components can interact. Composition $\times$ must be both *associative* and *commutative* in order to guarantee that different composable components may be assembled together in any order.

Consider the component $M_2$, defined as follows:

$$
M_2 : \begin{cases}
\text{variables:} & \begin{cases}
\text{inputs:} & x \\
\text{outputs:} & y
\end{cases} \\
\text{types:} & x, y \in \mathbb{R} \\
\text{behaviors:} & y = e^x
\end{cases}
$$

Component $M_2$ computes the value of the output variable $y$ as the exponential function of the input variable $x$. $M_1$ and $M_2$ are composable, since both common variables $x$ and $y$ have the same type, $x$ is an input variable to both $M_1$ and $M_2$, and the output variable $y$ of $M_2$ is fed as an input to $M_1$. It follows that their composition $M_1 \times M_2$ has a single input variable $x$, and computes the output $z$ as a function of $x$, that is $z = x/e^x$.

Now, consider component $M'_2$ that consists of an input variable $x$ and an output variable $z$, both of type real, where $z = \text{abs}(x)$ denotes the absolute value of $x$:

$$
M'_2 : \begin{cases}
\text{variables:} & \begin{cases}
\text{inputs:} & x \\
\text{outputs:} & z
\end{cases} \\
\text{types:} & x, z \in \mathbb{R} \\
\text{behaviors:} & z = \text{abs}(x)
\end{cases}
$$

Component $M'_2$ is not composable with $M_1$, because the two components share the same output variable $z$. Their composition is undefined, as it would result in conflicting rules for updating $z$.

We now lift the above concepts to contracts. The composition operator between two contracts, denoted by $\otimes$, shall be a partial function on contracts. Two contracts $\mathcal{C}$ and $\mathcal{C}'$ are composable if their variable types match. The resulting composition $\mathcal{C} \otimes \mathcal{C}'$ should specify, through its assumptions, the set of
environments in which these two components can interact. When $E_\mathcal{C} \otimes \mathcal{C}' \neq 0$, the composition $\mathcal{C} \otimes \mathcal{C}'$ is compatible: in this case, one usually says that the pair $(\mathcal{C}, \mathcal{C}')$ is compatible. By doing so, the resulting contract will expose how it should be used. Unlike component composability, contract compatibility is a combined syntactic and semantic property. Let us investigate this for our example. For $\mathcal{C}$ a contract, let $A_\mathcal{C}$ and $G_\mathcal{C}$ be its assumptions and guarantees and define

$$G_{\mathcal{C}_1 \otimes \mathcal{C}_2} = G_{\mathcal{C}_1} \land G_{\mathcal{C}_2}$$

$$A_{\mathcal{C}_1 \otimes \mathcal{C}_2} = \text{weakest} \left\{ A \left| \begin{array}{l} A \land G_{\mathcal{C}_2} \Rightarrow A_{\mathcal{C}_1} \\ \text{and} \\ A \land G_{\mathcal{C}_1} \Rightarrow A_{\mathcal{C}_2} \end{array} \right. \right\}$$

(2.4)

Thus $A_{\mathcal{C}_1 \otimes \mathcal{C}_2}$ is the weakest assumption such that the two referred implications hold. Thus, this overall assumption will ensure that, when put in the context of a component implementing the second contract, then the assumption of the first contract will be met, and vice-versa. Since the two assumptions were ensuring consistency for each contract, the overall assumption will ensure that the resulting composition is consistent. This definition of the contract composition therefore meets our previously stated requirements. The two contracts $\mathcal{C}_1$ and $\mathcal{C}_2$ are called compatible if the assumption computed as in (2.4) differs from $f$, the “false” predicate.

Consider contracts $\mathcal{C}_2$ and $\mathcal{C}_2'$ that we define as follows:

$$\mathcal{C}_2 : \begin{cases} \text{variables:} & \begin{cases} \text{inputs:} & u \\ \text{outputs:} & x \end{cases} \\ \text{types:} & u, x \in \mathbb{R} \\ \text{assumptions:} & \tau \\ \text{guarantees:} & x > u \end{cases}$$

$$\mathcal{C}_2' : \begin{cases} \text{variables:} & \begin{cases} \text{inputs:} & v \\ \text{outputs:} & y \end{cases} \\ \text{types:} & v, y \in \mathbb{R} \\ \text{assumptions:} & \tau \\ \text{guarantees:} & y = -v \end{cases}$$

\(^1\)So our definition of compatibility, which may seem unusual at a first glance, induces the usual definition of this term.
\( C_2 \) specifies components that for any input value \( u \), generate some output \( x \) such that \( x > u \) and \( C'_2 \) specifies components that generate the value of the output variable \( y \) as the function \( y = -v \) of the input \( v \). Observe that both \( C_2 \) and \( C'_2 \) are consistent. A simple inspection shows that \( C_1 \) and \( C_2 \) can be composed and their composition yields:

\[
\begin{align*}
C_1 \otimes C_2 : & \quad \text{variables: } \{ \text{inputs: } u, y \} \\
& \quad \{ \text{outputs: } x, z \} \\
& \quad \text{types: } x, y, u, z \in \mathbb{R} \\
& \quad \text{assumptions: } y \neq 0 \\
& \quad \text{guarantees: } x > u \land z = x/y
\end{align*}
\]

\( C_1 \) and \( C'_2 \) can also be composed and their composition yields:

\[
\begin{align*}
C_1 \otimes C'_2 : & \quad \text{variables: } \{ \text{inputs: } v, x \} \\
& \quad \{ \text{outputs: } y, z \} \\
& \quad \text{types: } v, x, y, z \in \mathbb{R} \\
& \quad \text{assumptions: } v \neq 0 \\
& \quad \text{guarantees: } y = -v \land z = x/y
\end{align*}
\]

Both compositions possess a non-empty assumption, reflecting that the two pairs \((C_1, C_2)\) and \((C_1', C'_2)\) are compatible.

In our example, it holds that contract composition is associative and commutative, that is, compositions \( C_1 \otimes (C_2 \otimes C_3) \) and \((C_1 \otimes C_2) \otimes C_3\) result in equivalent contracts, as well as compositions \( C_1 \otimes C_2 \) and \( C'_2 \otimes C_1 \), thus providing support for incremental system integration. This result will follow from the results of Section 4 on the meta-theory.

**Contract Refinement and Independent Development:** In all vertical design processes, the notions of abstraction and refinement play a central role. The concept of contract refinement must ensure the following: if contract \( C' \) refines contract \( C \), then any implementation of \( C' \) should 1) implement \( C \) and, 2) be able to operate in any environment for \( C \). Hence the following definition for the refinement pre-order \( \preceq \) between contracts: we say that the contract \( C' \) refines the contract \( C \), if \( E_{C'} \supseteq E_{C} \) and \( M_{C'} \subseteq M_{C} \). Since \( \preceq \) is a pre-order, refinement is a transitive relation. For our current series of
examples, and using previous notations, \( C' \leq C \) amounts to requiring that 1) \( A_C \) implies \( A_{C'} \), and 2) \( A_{C'} \Rightarrow G_{C'} \) implies \( A_C \Rightarrow G_C \).

Also, for all contracts \( C_1, C_2, C'_1 \) and \( C'_2 \), if \( C_1 \) is compatible with \( C_2 \) and \( C'_1 \leq C_1 \) and \( C'_2 \leq C_2 \), then \( C'_1 \) is compatible with \( C'_2 \) and \( C'_1 \otimes C'_2 \leq C_1 \otimes C_2 \).

We now illustrate this on our toy example, where we start with very abstract requirements for a component that implements a function \( z = x/e^x \).

Consider contracts \( C''_1 \) and \( C''_2 \), that we define as follows:

\[
C''_1 : \begin{cases}
\text{variables:} & \{ \text{inputs: } x, y \text{ outputs: } z \} \\
\text{types:} & x, y, z \in \mathbb{R} \\
\text{assumptions:} & (x > 0) \land (y \neq 0) \\
\text{guarantees:} & z \in \mathbb{R}
\end{cases}
\]

\[
C''_2 : \begin{cases}
\text{variables:} & \{ \text{inputs: } x \text{ outputs: } y \} \\
\text{types:} & x, y \in \mathbb{R} \\
\text{assumptions:} & \tau \\
\text{guarantees:} & y > 0
\end{cases}
\]

The contract \( C''_1 \) formalizes the most crude and abstract requirements for a divider. It requires that the denominator value (input variable \( y \)) is not equal to 0, and only ensures that the output value of \( z \) is some real number. The contract \( C''_2 \) specifies components that have an input variable \( x \) and an output variable of type real. The only requirement on the behavior of \( C''_2 \) is that \( y \) is strictly greater than 0. The composition \( C''_1 \otimes C''_2 \) is well defined. The contract \( C_1 \) refines \( C''_1 \), since it allows more inputs (its assumptions are weaker) and restricts the behavior of the output variable \( z \), by defining its behavior as the division \( x/y \). It follows that \( C_1 \) is also compatible with \( C''_2 \) and that \( C_1 \otimes C''_2 \leq C''_1 \otimes C''_2 \).

Finally, we have that \( M_1 \) and \( M_2 \) are implementations of their respective contracts. It follows that \( M_1 \times M_2 \) implements \( C_1 \otimes C''_2 \).

**Contract Conjunction and Viewpoint Fusion:** We now introduce the *conjunction* operator between contracts, denoted by the symbol \( \land \). Conjunction complements composition:
2.2. A primer on contracts

1. In the early stages of design, the system-level specification consists of a requirements document that is a conjunction of entailments assumptions ⇒ guarantees;

2. Full specification of a component can be a conjunction of multiple viewpoints, each covering a specific (functional, timing, safety etc.) aspect of the intended design and specified by an individual contract.

3. Conjunction supports reuse of a component in different parts of a design.

We state the desired properties of the conjunction operator as follows: Let $C_1$ and $C_2$ be two contracts. Then, $C_1 \land C_2 \preceq C_1$ and $C_1 \land C_2 \preceq C_2$, and for all contracts $C$, if $C \preceq C_1$ and $C \preceq C_2$, then $C \preceq C_1 \land C_2$.

To illustrate the conjunction operator, we consider a contract $C_{\tau}^1$ that specifies the timing behavior associated with $C_1$. For this contract, we introduce additional ports that allow us to specify the arrival time of each signal.

$$C_{\tau}^1 : \begin{cases} \text{variables:} & \{ \text{inputs:} \ t_x, t_y \ \\ \text{outputs:} \ t_z \} \\
\text{types:} & t_x, t_y, t_z \in \mathbb{R}_+ \\
\text{assumptions:} & \tau \\
\text{guarantees:} & t_z \leq \max(t_x, t_y) + 1 \end{cases}$$

where the symbol $\leq$ used in the guarantees denotes the usual order on real numbers. The contract $C_{\tau}^1$ is consistent with $C_1$, meaning that $C_{\tau}^1 \land C_1$ possesses implementations. Their conjunction $C_1 \land C_{\tau}^1$ yields a contract that guarantees, in addition to $C_1$ itself, a latency with bound 1 ms for it. Because there are no assumptions, this timing contract specifies the same latency bound also for handling the illegal input $y = 0$. In fact, the contract says more: because it does not mention the input $y$, it assumes any value of $y$ is acceptable. As a result, the conjunction inherits the weakest $\tau$ assumption of the timing contract, and cancels the assumption of $C_1$.

This, however, is clearly not the intent, since the timing contract is not concerned with the values of the signals, and is a manifestation of the weakness of this simple contract framework in dealing with contracts with different alphabets of ports and variables. We will further explain this aspect, and show how to address this problem, in Section 5. For the moment, we can fix
the problem by introducing $y$ in the interface of the contract, and use it in the assumptions, as in the following contract $C^T_2$

\[
\begin{cases}
\text{variables:} & \begin{cases}
\text{inputs:} & y, t_x, t_y \\
\text{outputs:} & t_z
\end{cases} \\
\text{types:} & y \in \mathbb{R}; t_x, t_y, t_z \in \mathbb{R}_+
\end{cases}
\]

assumptions: $y \neq 0$

 guarantees: $t_z \leq \max(t_x, t_y) + 1$

Note that this timing contract does not specify any bound for handling the illegal input $y = 0$, since the promise is not enforced outside the assumptions.

So far this example was extremely simple. In particular, it was stateless. Extension of this kind of Assume/Guarantee contracts to stateful contracts will be indeed fully developed in the coming sections and particularly in Section 5 and subsequent ones.
Having collected the “requirements” on contract theories, it is now timely to confront these to the previous work related to the broad concept of contract. This bibliographical note is limited to the grounding work on contract based design, across the different communities that have considered the problem, namely: software engineering, language design, system engineering, and formal methods in a broad sense. We report here a partial and limited overview of how this paradigm has been tackled in these different communities. While we do not claim being exhaustive, we hope that the reader will find her way to the different literatures. This note is organized into two parts. In the second part we focus on the development of contract based design for embedded systems and cyber-physical systems, which is the main focus of this work. In a first part, we review the work done by the other communities, under the generic name of “software engineering”. A more extensive and deeper coverage is given in subsequent bibliographical notes, for the different subtopics discussed in the different sections.

### 3.1 Contracts in software engineering

This part of the bibliographical note was inspired in part by the report [233].
Design by Contract is a software engineering technique popularized by Bertrand Meyer [200, 201] following earlier ideas from Floyd-Hoare logic [234, 155]. Imperative programs without (and then with) concurrency via shared variables are addressed [195]. Such programs can be seen as performing a sequence of actions reading inputs, transforming program states, and emitting outputs, where a state is the assignment of a value to all variables of the program. With reference to our previous nomenclature, such programs can be seen as our underlying class of “components”. On top of these, contracts (more often termed specifications in the VDM [163] community) can be considered to reason about programs.

### 3.1.1 Sequential imperative programs

Consider first the imperative programs with no concurrency. Floyd-Hoare logic assigns meaning to sequential imperative programs in the form of Hoare triples \((p, M, q)\) consisting of a precondition \(p\) on program states and inputs, a program command \(M\), and a postcondition \(q\) on program states and outputs. Regard the pair \(\mathcal{C} = \text{def} (p, q)\) as a contract and say that \(M\) implements \(\mathcal{C}\), written \(M \models \mathcal{C}\), if, when \(M\) operates from a state satisfying pre-condition \(p\), the resulting new state satisfies post-condition \(q\).

Hoare triples can be used to reason about the meaning of imperative programs with no concurrency. In particular, substitutability of specifications can be dealt with through a proper notion of refinement: Say that \(\mathcal{C}' \preceq \mathcal{C}\), written \(\mathcal{C}' \preceq \mathcal{C}\), if, for every command \(M\), \(M \models \mathcal{C}'\) implies \(M \models \mathcal{C}\). Sufficient conditions expressed on the elements of the Hoare triple were proposed that ensure refinement in the above sense. One such example of sufficient condition for \(\mathcal{C}' \preceq \mathcal{C}\) is \(p \Rightarrow p'\) and \(q \Rightarrow q\)—note the opposite implications. When the command \(M\) itself contains states which are preserved from one execution to the next, then additional invariant conditions \(g\) can be added that constrain the possible states. These could be considered as post-conditions which must be true irrespective of the pre-conditions. Therefore, they must be preserved or strengthened through refinement.

This notion is made more explicit in the design-by-contract technique implemented in the object-oriented language Eiffel, proposed by Meyer [201]. The general idea follows that of Hoare triples. Every function of a class (there

\(^1\)Which, according to our philosophy, we regard as a “component”.

Positioning of this Monograph and Bibliographical Note
called features and queries) can be augmented with conditions, called pre-
conditions, that the caller must satisfy in order for the call to be considered
correct. These conditions are expressed in terms of assertions that constrain
the value of variables in their scope, including the formal arguments of the
functions and the state of the class, intended as the collection of the values
of its variables. Thus, for instance, one can require that a function be called
with a specific argument range. The caller must ensure that the precondition
is true before executing the call. Conversely, the programmer can specify a
condition that must be true of the state of the object or of the return value after
the execution of the function, in the form of a postcondition. The postcondi-
tion can refer also to the value of the state before the call (the old value), so
that assertions can capture the side effects of executing the function. Finally,
a class invariant states properties that must be true of the state of the class at
all times (except during the execution of a function, in which the state can
temporarily violate the invariants). Contracts work through derived classes as
expected: preconditions are weakened while postconditions and invariants are
strengthened, so that an implementation of a class can safely be used in place
of its specification. Besides functions, Eiffel can also express loop invariants
and loop variants, which are extremely useful in program verification.

Contracts in Eiffel can be compiled with the program, so that the validity
of the assertions is checked during execution. Otherwise, no other verification
method is implemented natively in the language, but formal proofs must be
established in other ways. Nevertheless, the presence of these constructs in
the language is extremely important from a methodological point of view.

Compared to other contract methodologies presented in this work, the
contracts in Eiffel are much more restricted in terms of their expressiveness.
The conditions are mostly static assertions on the values of the variables, and
say little about the dynamic interaction of the functions or their behavior over
time. For instance, one cannot express the fact that a function must be exe-
cuted before another (a kind of “behavioral” property), unless some dedicated
variable is artificially added to represent previous executions and is properly
updated. In other words, one would need to overlay a state machine model
in order to represent the explicit object evolution, in a form similar to many
of the interface and contract-based methods discussed in this work. It is un-
clear, however, how this approach could work through derived classes, since
Positioning of this Monograph and Bibliographical Note

inheritance, an essential notion in object oriented programming, would not impose any form of refinement on the state machines. The language is also not concerned with aspects of concurrency and there is no explicit notion of composition of contracts. Consequently, the approach does not discuss issues of compatibility or consistency. One can instead think of multiple inheritance as a form of conjunction, although implementations do not need to satisfy exact boundary conditions (they are lower bounds, but not necessarily greatest lower bounds, and therefore not real conjunctions). In summary, contracts in Eiffel are a form of advanced structured assertions which are automatically checked at run time and propagated through inheritance, but additional tools and methods are required to make full use of their expressive power.

With the aim of addressing service oriented architectures, a multiple layering was proposed for Meyer’s contracts by Beugnard et al. [51]. The basic layer specifies operations, their inputs, outputs and possible exceptions. The behavior layer describes the abstract behavior of operations in terms of their preconditions and postconditions. The third layer, synchronisation, corresponds to real-time scheduling of component interaction and message passing. The fourth, quality of service (QoS) level, details non-functional aspects of operations. The contracts proposed by Beugnard et al. are subscribed prior to service invocation and may also be altered at runtime, thus extending the use of contracts to Systems of Systems [203].

The SCOOP (Simple Concurrent Object-Oriented Programming) model was proposed by B. Meyer to extend Eiffel to concurrency [72]. SCOOP uses the Rely/Guarantee paradigm that we review in the next section.

3.1.2 Imperative programs with concurrency

So far contracts consisting of pre/postconditions naturally fit imperative sequential programming. In situations where such imperative programs may operate concurrently, interference on shared variables can occur. Rely/Guarantee rules [159, 163] were thus added to interface contracts.

Rely conditions state assumptions about any interference on shared variables during the execution of operations by the system’s environment. Guarantee conditions state obligations of the operation regarding shared variables.

http://cme.ethz.ch/research/
Thus, a contract is now a richer pair $C = ([p, r], [g, q])$, consisting of a precondition $p$, a rely condition $r$, a guarantee condition $g$, and a post-condition $q$. Pre- and post-conditions are as before, but rely and guarantee conditions are new notions to handle concurrency: $r$ specifies what is expected from the environment of the program and, assuming that the environment behaves as expected, $g$ specifies what the effect on the environment will be of applying the considered program. Say that a program command $M$ implements $C$ if, when $M$ operates on a state satisfying $p$ under an environment satisfying $r$, then the resulting new state will satisfy $q$ and the effect on the environment will satisfy $g$. Again, refinement of contracts can be considered: say that $C' \preceq C$ if: 1) every environment that is acceptable for $C$ is also acceptable for $C'$, and 2) every command $M$ that implements $C'$ in an environment satisfying the expectations of $C$ also implements $C$. Proof rules exist for the parallel composition of program commands, see [162]. A precis of rely/guarantee reasoning is available in the Appendix of [162].

We devote Chapter 7 of this monograph to the study of Rely/Guarantee reasoning with our perspective. In particular we exhibit a tight relation with Assume/Guarantee contracts.

### 3.1.3 Contracts in Model Driven Engineering

The concepts of interface and contract were subsequently further developed in the Model Driven Engineering (MDE) community [169, 237, 187]. 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) [209, 255]. 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. OCL can, for instance, be used to specify an invariant in terms of the conditions that must be satisfied before and after the execution of a method or the step of a state machine, providing ways to express assumptions and guarantees. In parallel to MDE, the Platform-based design (PBD) approach [114, 221] lends itself naturally to
the use of contracts, where a functional specification relies (makes assumptions) on the provision (the guarantees) of an architecture, in what are known as vertical contracts [236]. Similarly to OCL, constraints can be used to express both the coordination between the levels of abstractions, and conditions on execution [24] using explicit quantities to represent non-functional properties [100]. Likewise, attributes on port methods have been used to represent non-functional requirements or provisions of a component [73]. 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 — actually, several MDE related tools, such as GME and Rational Rose, automatically generate executable code from this specification [28, 188, 223]. The notion of refinement is replaced by the concept of class inheritance. From a contract theory point of view, this approach has several limitations. Inheritance, for instance, does not properly cover aspects related to behavior refinement, in the sense that the abstract class is unable to constrain the actions that its implementations may perform, and is instead limited to establishing the signature of the methods. Nor is it made precise what it means to take the conjunction of interfaces, which can only be approximated by multiple inheritance, or to compose them.

In a continuing effort since his joint work with W. Damm on Live Sequence Charts (LSC) in 2000 [98] with its Play-Engine implementation [149], David Harel has developed the concept of behavioral programming [150, 148, 151], which puts in the forefront scenarios as a program development paradigm — not just a specification formalism. In behavioral programming, b-threads generate a flow of events via an enhanced publish/subscribe protocol. Each b-thread is a procedure that runs in parallel to the other b-threads. When a b-thread reaches a point that requires synchronization, it waits until all other b-threads reach synchronization points in their own flow. At synchronization points, each b-thread specifies three sets of events: requested events: the thread proposes that these be considered for triggering, and asks to be notified when any of them occurs; waited-for events: the thread does not request these, but asks to be notified when any of them is triggered; and blocked events: the thread currently forbids triggering any of these events. When all b-threads are at a synchronization point, an event is chosen (according to some policy), that is requested by at least
one b-thread and is not blocked by any b-thread. The selected event is then triggered by resuming all the b-threads that either requested it or are waiting for it. This mechanism was implemented on top of Java and LSCs. The execution engine uses planning and model checking techniques to prevent the system from falling into deadlock, where all requested events are blocked. Behavioral programming is incremental in that new threads can be added to an existing program without the need for making any change to this original program: new deadlocks that are created by doing so are pruned away by the execution engine. While behavioral programming cannot be seen as a paradigm of contracts, it shares with contracts the objectives of incremental design and declarative style of specification.

3.2 Contracts for (possibly cyber-physical) systems

The frameworks of contracts developed in the area of Software Engineering were established as useful paradigms for component based software system development. In this monograph, we also (and in fact mainly) target the area of computer controlled systems, more recently referred to as cyber-physical systems, where reactive systems [146, 152, 142, 196] are encountered. Reactive systems are systems that continuously interact with some environment, as opposed to transformational systems [142], considered in Object-Oriented programming. For reactive systems, model-based development (MBD) is generally accepted as a key enabler due to its capabilities to support early validation and virtual system integration. MBD-inspired design languages and tools include SysML [208] or AADL [211] for system level modeling, Modelica [133] for physical system modeling, Matlab-Simulink [168] for control-law design, and Scade [206, 47] and TargetLink for detailed software design. UML-related standardization efforts in this area also include the MARTE<sup>3</sup> [http://www.omg.org/omgmarte/ UML profile for real-time systems.

As a first difference with the context of Software Engineering, we are dealing here with systems that truly operate in parallel — sensors, devices, plant, and actuators operate in parallel with the computer system on board that controls all of them. Threads with shared variable communication are

Positioning of this Monograph and Bibliographical Note

not the natural paradigm for this kind of parallelism. The systems we consider interact via input/output communication such as encountered in block- or dataflow-diagrams (Matlab-Simulink, Scade, SysML), or even via non oriented exchange of communicated variables (Modelica). If we restrict ourselves to discrete time dynamics, the adequate paradigm is that of Transition Systems with parallel composition via intersection. While this form of concurrency is less familiar to Software Engineers, it is indeed much simpler from the mathematical standpoint. The parallel composition (of block- or dataflow-diagrams) is commutative and associative. Associativity formalizes the desirable property that systems can be rearchitected by degrouping subsystems and regrouping them differently.

Contract theories for this context were considered in the community of formal verification. They were initially developed as specification formalisms able to refuse certain inputs from the environment. Abadi and Lamport (with Wolper for the first publication) [5, 3] were the first to propose a comprehensive Assume/Guarantee specification theory for Transition Systems. The first publication introduced the game point of view in distinguishing component from environment and using this for defining refinement. The second long publication proposed a comprehensive specification framework with Assumptions (restricted to safety properties) and Guarantees (both safety and liveness are covered). The composition of specifications is studied and the issue of circular reasoning is pinpointed and solved by a reinforcement of the assumptions of the composition, with comparison to the “intuitive” definition for them. Dill proposed asynchronous trace structures with failure behaviors [116]. A trace structure is a representation of a component or interface with two sets of behaviors. The set of successes are those behaviors which are acceptable and guaranteed by the component. Conversely, the set of failures are behaviors which drive the component into unacceptable states, and are therefore refused. This work focuses primarily on the problem of checking refinement, and does not explore further the potentials of the formalism from a methodological point of view. The work by Dill was later extended by Wolf in the direction of synchronous systems [256]. Negulescu later generalizes the algebra to Process Spaces which abstract away the specifics of the behaviors, and derives new composition operators [205]. This particular abstraction technique was earlier introduced by Burch with Trace Algebras to construct
3.2. *Contracts for (possibly cyber-physical) systems*

conservative approximations [67], and later generalized by Passerone and Burch [68, 214] to study generic trace structures with failure behaviors and to formalize the problem of computing the quotient (there called mirror) [212]. Methodological aspects of contract-based design of Cyber-Physical Systems are extensively discussed in [236]. This monograph aims at proposing, for model based design of systems and CPS, a new vista on contracts. In the next section, we propose an all encompassing meta-theory of contracts.
In Section 2 and its bibliographical discussion, we showed that a number of frameworks — specification, interface, contract theories, and more — were proposed to cope with the issues of system development in a supplier chain. The list of such frameworks will be further increased in the forthcoming sections. This calls for developing a “birds-eye view” of the subject, by which the essence and commonalities of such frameworks will be highlighted. In software engineering, meta-models are “models of models”, i.e., formal ways of specifying a certain family of models [235, 215, 27]. Analogously, we call *meta-theory* a way to specify a particular family of theories. In this section we propose a meta-theory of contracts.

In this meta-theory we will mainly focus on semantic concepts. Clearly, components, contracts, and the associated relations and operators, must be expressed in some language which defines the syntax for specifying them. The properties of the specification language are important in several respects. In particular, the finite nature of the representation may limit the kind of objects that can be represented, and could consequently affect the realizability (closure) of the operators of our theory — we will pay special attention to this issue. Nonetheless, in this section on the “meta-theory”, we are interested
4.1. Components

primarily in the relations between the objects that the language describes, irrespective of how they are represented. Therefore, we will proceed under the hypothesis that questions of representations have been adequately addressed. Hence, when referring to components and contracts, we implicitly assume that they are described in some language whose semantics maps to the concepts that we present in this section. How this meta-theory can be instantiated to various concrete frameworks is discussed in subsequent sections.

The meta-theory we develop here is novel. There are very few attempts of that kind. In fact, the only ones we are aware of are the recent works by Bauer et al. [29] and Chen et al. [85], which follow a different (and complementary) approach. The discussion of this work is deferred to the bibliographical Section 4.8.

Our meta-theory is summarized in Tables 4.1 and 4.2. It comes as a few primitive concepts and Axioms, on top of which derived concepts can be built. A number of key Properties can be proved about the resulting framework. These Axioms and Properties are developed in the course of this section. They demonstrate that contracts are a convenient paradigm to support incremental development and independent implementability in system design.

4.1 Components

To introduce our meta-theory, we start from a universe $\mathcal{M}$ of possible components, each denoted by the symbol $M$ or $E$, and a universe of their specifications, or contracts, each denoted by the symbol $\mathcal{C}$. Our meta-theory does not presume any particular modeling style, neither for components nor for contracts — we have seen in Section 2 an example of a (very simple) framework for static systems. More generally, some frameworks may represent components and contracts with sets of discrete time or even continuous time traces, other theories use logics, or state-based models of various kinds, and so on.

We assume a composition operator $M_1 \times M_2$ acting on pairs of components. Component composition $\times$ is partially, not totally, defined. Two components such that $M_1 \times M_2$ is well defined are called composable. Composability of components is meant to be a typing property. For convenience, we augment $\mathcal{M}$ with an extra element $\star \notin \mathcal{M}$, where $\star$ means “undefined” and we make $\times$ total by setting $M_1 \times M_2 = \star$ if $M_1$ and $M_2$ are not composable,
Table 4.1: Summary of the meta-theory of contracts. We first list \textit{primitive} concepts and then \textit{derived} concepts introduced by the meta-theory. $\mathcal{C}_1$, $\mathcal{C}_2$, and $\mathcal{C}$ are implicitly universally quantified over some underlying class $\mathcal{C}$ of contracts which depends on the particular contract framework considered.

<table>
<thead>
<tr>
<th>Concept</th>
<th>Definition and generic properties</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Primitive</strong></td>
<td></td>
</tr>
<tr>
<td>Component</td>
<td>$\mathcal{M}$ is an underlying set of components; elements of $\mathcal{M}$ are denoted by $M$</td>
</tr>
<tr>
<td>Composability of components</td>
<td>A type property on pairs of components $(M_1, M_2)$</td>
</tr>
<tr>
<td>Composition of components</td>
<td>$M_1 \times M_2$ is defined if and only if $M_1$ and $M_2$ are composable; It is required that $\times$ is associative and commutative; By convention, we augment $\mathcal{M}$ with some extra element $\star$ (&quot;undefined&quot;) and extend $\times$ by setting $M_1 \times M_2 = \star$ if $M_1$ and $M_2$ are not composable</td>
</tr>
<tr>
<td>Environment</td>
<td>An \textit{environment} for component $M$ is a component $E$ such that $E \times M$ is defined</td>
</tr>
<tr>
<td><strong>Derived</strong></td>
<td></td>
</tr>
<tr>
<td>Contract</td>
<td>We assume a class $\mathcal{C}$ of \textit{contracts} $\mathcal{C}$ whose semantics is a pair $(E_\mathcal{C}, M_\mathcal{C})$, where $M_\mathcal{C} \subseteq \mathcal{M}$ is a subset of implementations and $E_\mathcal{C} \subseteq \mathcal{M}$ a subset of environments such that every pair $(E, M) \in E_\mathcal{C} \times M_\mathcal{C}$ is composable</td>
</tr>
<tr>
<td>Consistency</td>
<td>$\mathcal{C}$ is \textit{consistent} iff it has at least one implementation: $M_\mathcal{C} \neq \emptyset$</td>
</tr>
<tr>
<td>Compatibility</td>
<td>$\mathcal{C}$ is \textit{compatible} iff it has at least one environment: $E_\mathcal{C} \neq \emptyset$</td>
</tr>
<tr>
<td>Implementation</td>
<td>$M \models^\mathcal{C} \mathcal{E}$ if and only if $M \in M_\mathcal{E}$</td>
</tr>
<tr>
<td></td>
<td>$E \models^\mathcal{C} \mathcal{E}$ if and only if $E \in E_\mathcal{E}$</td>
</tr>
</tbody>
</table>

and $\star \times \ldots = \ldots \times \star = \star$ where $\ldots$ is an arbitrary element of $\mathcal{M} \cup \{\star\}$ — this extension is only for technical convenience in order to make subsequent definitions shorter.

In order to guarantee that different composable components may be assembled together in any order, it is required that component composition $\times$
4.2. Contracts

Table 4.2: Summary of the meta-theory of contracts, continued. This table points to the Axioms and Properties satisfied by the different notions as will be detailed later in the section.

<table>
<thead>
<tr>
<th>Refinement</th>
<th>$C' \leq C$ iff $E_{C'} \supseteq E_C$ and $M_{C'} \subseteq M_C$; <strong>Property</strong> 1 holds</th>
</tr>
</thead>
<tbody>
<tr>
<td>GLB and LUB of contracts</td>
<td>$C_1 \wedge C_2 = \text{Greatest Lower Bound (GLB) of } C_1 \text{ and } C_2 \text{ for } \preceq$; $C_1 \vee C_2 = \text{Least Upper Bound (LUB) of } C_1 \text{ and } C_2 \text{ for } \preceq$; <strong>Axiom</strong> 1 is in force and <strong>Property</strong> 2 holds Say that $(C_1, C_2)$ is a <strong>consistent pair</strong> if $C_1 \wedge C_2$ is consistent</td>
</tr>
</tbody>
</table>
| Composition of contracts | $C_1 \otimes C_2$ is defined if $\forall M_1 \models^u C_1, \forall M_2 \models^u C_2$, then $(M_1, M_2)$ is composable and $C_1 \otimes C_2 = \min \left\{ C \mid \begin{array}{l} \forall M_1 \models^u C_1 \\
\forall M_2 \models^u C_2 \\
\forall E \models^e C \\
M_1 \times M_2 \models^u C \\
E \times M_2 \models^u C_1 \\
E \times M_1 \models^u C_2 \end{array} \right\}$ **Axiom** 2 is in force; **Properties** 3, 4, 5, and 6 hold; Say that $(C_1, C_2)$ is a **compatible pair** if $C_1 \otimes C_2$ is compatible |
| Quotient | $C_1 / C_2 = \max\{ C \mid C \otimes C_2 \preceq C_1 \}$; **Axiom** 3 is in force and **Property** 7 holds |

is associative and commutative. Data-flow of block-diagram graphical formalisms, for which composition is by wiring selected ports from different blocks, satisfy the above property — Simulink, Scade, and Modelica, are instances of such formalisms. An **environment** for a component $M$ is another component $E$ composable with $M$.

4.2 Contracts

In our primer of Section 2, we have highlighted the importance of the valid environments associated with contracts, for which an implementation will operate satisfactorily. At the abstract level of the meta-theory, we make this
explicit next. In the sequel, for $X$ a set, we denote its powerset by

$$\phi(X).$$

**Definition 4.1.** We assume a class $C$ of contracts $\mathcal{C}$ whose semantics is a pair $\llbracket C \rrbracket = (E_\mathcal{C}, M_\mathcal{C}) \in \phi(\mathcal{M}) \times \phi(\mathcal{M})$, where:

- $M_\mathcal{C} \subseteq \mathcal{M}$ is the set of implementations of $\mathcal{C}$, and
- $E_\mathcal{C} \subseteq \mathcal{M}$ is the set of environments of $\mathcal{C}$.
- For any $(E, M) \in E_\mathcal{C} \times M_\mathcal{C}$, $E$ is an environment for $M$.

A contract having no implementation is called *inconsistent*. A contract having no environment is called *incompatible*. Write

$$M \models^u \mathcal{C} \text{ and } E \models^e \mathcal{C}$$

to express that $M \in M_\mathcal{C}$ and $E \in E_\mathcal{C}$, respectively.

---

**Comment 4.1.** In the meta-theory the class $C$ is abstract. Each particular contract framework comes with a concrete definition of $C$ based on a concrete syntax, and instantiates all the concepts listed in the last column of Tables 4.1 and 4.2, thus making them effective (amenable of algorithmic manipulations). In turn, not all pairs $(E_\mathcal{C}, M_\mathcal{C}) \in \phi(\mathcal{M}) \times \phi(\mathcal{M})$ can be the semantics of a contract of class $C$. While the definition of contract consistency complies with the intuition, the definition of contract compatibility may seem strange at first glance. We shall, however, see that it specializes to known notions for concrete theories.

---

### 4.3 Refinement and conjunction

To support independent implementability, the concept of contract refinement must ensure the following: if contract $\mathcal{C}'$ refines contract $\mathcal{C}$, then any implementation of $\mathcal{C}'$ should implement $\mathcal{C}$ and be able to operate in any environment for $\mathcal{C}$. Hence the following definition for the refinement preorder $\preceq$...
between contracts: \( C' \) refines \( C \), written \( C' \leq C \), if and only if \( M_{C'} \subseteq M_C \) and \( E_{C'} \supseteq E_C \). As a direct consequence, the following property holds, which justifies the use of the term “refinement” for this relation:

**Property 1 (refinement).** Any implementation of \( C' \) is an implementation of \( C \). Any environment of \( C \) is an environment of \( C' \). □

At this point, since not all pairs \( (E_C, M_C) \in \wp(M_C) \times \wp(M_C) \) can be the semantics of a contract of class \( C \) (see Comment 4.1), we need the following axiom on the contract language:

**Axiom 1.** For any \( C' \subseteq C \):

1. The Greatest Lower Bound (GLB) \( \bigwedge C' \) exists in \( C \);

2. In addition, the existence of the Least Upper Bound (LUB) \( \bigvee C' \) in \( C \) is required for the quotient to exist.

Both GLB abd LUB refer to refinement order. □

Although strong, this axiom is satisfied by the instances of contract languages we know, see the subsequent sections for this. The LUB is only required for the existence of a quotient. For some instances of the meta-theory a quotient may not exist, in which case the existence of the LUB is not needed.

Axiom 1 allows us to define the conjunction \( C_1 \land C_2 \) of contracts \( C_1 \) and \( C_2 \) as the GLB of these two contracts. The intent is to define this conjunction as the intersection of sets of implementations and the union of sets of environments. However, not every pair of sets of components can be the semantics of a contract belonging to class \( C \). The best approximation consists in taking the greatest lower bound for the refinement relation. The following immediate properties hold:

**Property 2 (shared refinement).** Any contract that refines \( C_1 \land C_2 \) also refines \( C_1 \) and \( C_2 \). Any implementation of \( C_1 \land C_2 \) is a shared implementation of \( C_1 \) and \( C_2 \). Any environment of \( C_1 \) or \( C_2 \) is an environment of \( C_1 \land C_2 \). □

The conjunction operation formalizes the intuitive notion of a “set of contracts” or a “set of requirements”.
4.4 Parallel composition

On top of component composition, we define a contract composition $C_1 \otimes C_2$, whose intuition is as follows: composing two implementations of $C_1$ and $C_2$ should yield an implementation of $C_1 \otimes C_2$ and any environment for $C_1 \otimes C_2$, when composed with an implementation for $C_1$, should yield a valid environment for $C_2$ and vice-versa. Observe that $E \models^+ C$ implies that $E$ is composable with any implementation of $C$, and thus $E \times M_1$ and $E \times M_2$ are well defined.

The parallel composition of contracts has proved to be a difficult concept in many frameworks. For Assume/Guarantee contracts detailed in Section 5, the associativity of the contract composition was not proved — it was not even stated in the original reference [40]. For both Interface Automata and Modal Interfaces, the definition of the parallel composition is made difficult due to the mechanism of pruning for compatibility, as we shall see in Section 8. Several references reported the correction of errors in previous definitions: errors in [177] were reported in [228] and errors in [228, 227] were reported and corrected in [62]. Failure of previous definitions of the parallel composition to be associative were reported in [66, 62]. Our meta-theory provides important clarifications. Our definition of the parallel composition is closely driven by the objective of independent implementation and its definition is very natural in this respect. We show that contract composition is not associative in general, but only sub-associative, a weakening of associativity that involves refinement for its definition. We explain that sub-associativity is indeed sufficient to enable independent development. We provide a tight sufficient condition for associativity. The bottom line is that our meta-theory clarifies the notion of parallel composition, for both Assume/Guarantee contracts and interface theories.

Since the parallel composition of contracts is not associative in general, the definition stated in Tables 4.1 and 4.2 is not sufficient and we must directly define the contract composition for any arity. Let $I$ be a finite set and $C_i, i \in I$ be a set of contracts indexed by $I$. The parallel composition of this set of contracts is defined if and only if any set of implementations $\{M_i \mid M_i \models^w C_i, i \in I\}$ is composable. Then:

$$\bigotimes_{i \in I} C_i \overset{\text{def}}{=} \min C_I,$$  \hspace{1cm} (4.1)
where $C_I$ is the set of contracts $C$ such that the following implication holds for every tuple $M_i, i \in I$ of components and every environment $E$:

$$
\begin{bmatrix}
M_i \models^w C_i \\
E \models^w \forall C
\end{bmatrix} \Rightarrow \begin{bmatrix}
\left( \times_{j \in I} M_j \right) \models^w \forall C
\end{bmatrix}
$$

(4.2)

In (4.1) “min” refers to the refinement order. In the right hand side of (4.2), the implication $[\ldots] \Rightarrow [\ldots]$ is under the scope of the quantification $\forall M_i, i \in I, \forall E$, which means “for every tuple $M_i, i \in I$ and $E$, the implication $[\ldots] \Rightarrow [\ldots]$ holds. For this to make sense, we assume the following:

**Axiom 2.** The min in (4.1) exists and is unique.

Axiom 2 rewrites $\land C_i \in C_I$ and thus $\bigotimes_{i \in I} C_i = \land C_I$. At this point we need to clarify what happens if some of the preconditions of the implication in (4.2) are vacuous. For $C$ fixed, if no $M_i$ exists that implements $C_i$, then the properties involving $M_i$ on the right hand side are vacuously satisfied. Note that $\bigotimes_{i \in I} C_i$ is incompatible if and only if the set $C_I$ contains only incompatible contracts $C$, i.e., such that no $E$ exists for which $E \models^w C$.

To conform to the usage, we say that $C_i, i \in I$ are compatible contracts if their composition is defined and compatible in the sense of Tables 4.1 and 4.2. The following property follows directly from the definition of the contract composition:

**Property 3 (independent implementability).** Compatible contracts can be independently implemented.

The following lemma holds:

**Lemma 4.2.** Let contracts be such that $C'_i \preceq C_i$ for every $i \in I$. If $\bigotimes_{i \in I} C_i$ is well defined, then, so is $\bigotimes_{i \in I} C'_i$ and $C'_I \supseteq C_I$ holds, where $C'_I$ is defined by formula (4.2) with $C'_i$ in lieu of $C_i$.

**Proof.** Since $C_I$ is defined, it follows that every tuple $(M_i, i \in I)$ of respective implementations of these contracts is composable. Hence, $C'_I$ is defined too. Next, since $C'_i \preceq C_i$ for every $i \in I$, $M_i \models^w C'_i, i \in I$ implies $M_i \models^w C_i, i \in I$ and $E \times (\times_{j \neq i} M_j) \models^w C_i$ implies $E \times (\times_{j \neq i} M_j) \models^w C'_i$. Therefore, replacing, in the right hand side of (4.2), $C_i$ by $C'_i$ can only increase the set $C_I$. 

□
The following property is a direct corollary of Lemma 4.2:

**Property 4 (independent refinement).** For all contracts $C_i, i \in I$ and $C'_i, i \in I$, if $C_i, i \in I$ are compatible and $C'_i \preceq C_i, i \in I$ both hold, then $C'_i, i \in I$ are compatible and

$$\bigotimes_{i \in I} C'_i \preceq \bigotimes_{i \in I} C_i$$

Referring to the discussion of Section 2.1, Properties 3 and 4 are fundamental, particularly in top-down design, which consists in decomposing a system-level contract $C$ into subsystem contracts $C_i, i \in I$ for further independent development. To ensure that independent development will not lead to integration problems, it is enough to verify that $\bigotimes_{i \in I} C_i \preceq C$. Then, any $C_i$ can be independently refined into $C'_i$ and the composition $\bigotimes_{i \in I} C'_i$ will be a refinement of $C$.

In using contract based design, our recommendation is that, at each refinement step, the decomposition of a given contract into a parallel composition of subcontracts should not involve a large number of them. This will mitigate the risk of state explosion in the considered parallel composition.

This parallels compositional verification, in which $\times_{i \in I} M_i \models^w P$ is to be checked, where $M_i$ are detailed implementations, $P$ is a property, and $I$ may be a large set, thus giving rise to state explosion for $\times_{i \in I} M_i$. Techniques similar to our use of stepwise refinement have been proposed to verify such properties in an incremental way [244, 90, 139, 4, 170].

**Property 5 (commutativity, sub-associativity).** For $C_i, i = 1, \ldots, n$ any finite set of contracts, we have:

$$C_1 \otimes C_2 = C_2 \otimes C_1 \quad (4.3)$$

$$\bigotimes_{1 \leq i \leq n} C_i \preceq \left(\bigotimes_{1 \leq i < n} C_i\right) \otimes C_n \quad (4.4)$$

The precise meaning of these formulas is: if one side of formula (4.3) or (4.4) is defined, then the other side of this formula is defined as well and the two sides are related as indicated.

**Proof.** The commutativity property (4.3) is a direct consequence of Axiom 2, formulas (4.2) and (4.1), and the commutativity of component composition. Thus we focus on the associativity. For readability issues, we give the proof for $n = 3$ but the proof for the general case is the same except for heavier notations due to the use of $n$-ary operations instead of binary ones.
To shorten notations, we reflect in its index the parenthesizing in a parallel composition expression, that is, we write $\mathcal{C}_{12}$ instead of $\mathcal{C}_1 \otimes \mathcal{C}_2$, $\mathcal{C}_{123}$ instead of $\mathcal{C}_1 \otimes \mathcal{C}_2 \otimes \mathcal{C}_3$ and $\mathcal{C}_{(12)3}$ instead of $(\mathcal{C}_1 \otimes \mathcal{C}_2) \otimes \mathcal{C}_3$.

Recall that, as a prerequisite of the contract composition $\mathcal{C}_1 \otimes \mathcal{C}_2$, any composition $M_1 \times M_2$ of implementations of $\mathcal{C}_1$ and $\mathcal{C}_2$ is defined (i.e., different from $\star$). Then, in writing the formulas below, we recall that the condition $E \times M_1 \models^e \mathcal{C}_2$ implies in particular that $E \times M_1$ is defined, see the comment following Definition 4.1.

By Axiom 2 and formulas (4.2) and (4.1), $\mathcal{C}_{12}$ is the unique minimum among the contracts satisfying the following property:

$$
\begin{align*}
\forall M_1 \quad M_1 &\models^m \mathcal{C}_1 \\
\forall M_2 \quad M_2 &\models^m \mathcal{C}_2 \\
\forall E &\models^e \mathcal{C}_{12}
\end{align*}
\Rightarrow
\begin{align*}
M_1 \times M_2 &\models^m \mathcal{C}_{12} \\
E \times M_1 &\models^e \mathcal{C}_2 \\
E \times M_2 &\models^e \mathcal{C}_1
\end{align*}
$$

(4.5)

For the same reasons, $\mathcal{C}_{(12)3}$ is the unique minimum among the contracts satisfying the following property:

$$
\begin{align*}
\forall M_{12} \quad M_{12} &\models^m \mathcal{C}_{12} \\
\forall M_3 &\models^m \mathcal{C}_3 \\
\forall E &\models^e \mathcal{C}_{(12)3}
\end{align*}
\Rightarrow
\begin{align*}
M_{12} \times M_3 &\models^m \mathcal{C}_{(12)3} \\
E \times M_{12} &\models^e \mathcal{C}_3 \\
E \times M_3 &\models^e \mathcal{C}_{12}
\end{align*}
$$

(4.6)

Using (4.5), we can select in (4.6) $M_{12} = M_1 \times M_2$ where $M_i \models^m \mathcal{C}_i$, $i = 1, 2$. This yields the following property, where all the written component compositions are defined and index $i$ ranges over the set $\{1, 2, 3\}$:

$$
\begin{align*}
M_i &\models^m \mathcal{C}_i \\
E &\models^e \mathcal{C}_{(12)3}
\end{align*}
\Rightarrow
\begin{align*}
M_1 \times M_2 \times M_3 &\models^m \mathcal{C}_{(12)3} \\
E \times (\times_{j \neq i} M_j) &\models^e \mathcal{C}_i
\end{align*}
$$

(4.7)

Thus, we proved that an implementation for $\mathcal{C}_{123}$ of the form $M_{123} = M_1 \times M_2 \times M_3$, is indeed an implementation for $\mathcal{C}_{(12)3}$. At the same time, we proved that an environment for $\mathcal{C}_{(12)3}$, $E$, is also an environment for all $\mathcal{C}_i$ when composed with the other implementations $j \neq i$, hence, by (4.2), it is also an environment for $\mathcal{C}_{123}$. This proves (4.4). $\square$

To address arbitrary parenthesizing, we encode the composition formula as its parse tree $T$, whose leaves are labeled by the set $\{1, \ldots, n\}$ and we
denote by $\bigotimes_T C_i$ the contract composition with this parenthesizing. Sub-associativity with this nested parenthesizing follows as a corollary of Property 5 by structural induction over the depth of $T$:

$$\bigotimes_{1 \leq i \leq n} C_i \leq \bigotimes_T C_i$$ (4.8)

**Comment 4.3** (sub-associativity enables independent implementability).

With reference to a SysML-like description of architectures, parentheses in the composition correspond to boxes of the considered architecture and the associativity ensures full freedom regarding how to assemble a subset of components as a box to form a subsystem. Associativity and commutativity allow for rearchitecturing. What about the weaker property of sub-associativity? For this discussion and its follow-up in Comment 4.5, the reader is referred to Figure 2.2. The system level contract $C$ is first refined by the OEM into an architecture $C_1 \otimes C_2 \otimes C_3$ and the subsystem level $C_i$ are delegated to different suppliers. Tier-1 supplier #1 returns an implementation $M_1$ for his subcontract. Tier-1 supplier #2 further refines its contract to an architecture $C_{21} \otimes C_{22}$ and delegates its implementation to tier-2 suppliers, which return $M_{21}$ and $M_{22}$ respectively. Tier-1 supplier #3 does the same. Finally, when performing system integration, the OEM gets $M_1 \times M_{21} \times M_{22} \times M_{31} \times M_{32}$. The latter is, by definition, a correct implementation of $C_1 \otimes (C_{21} \otimes C_{22}) \otimes C_{31} \otimes C_{32} \leq C_1 \otimes (C_{21} \otimes C_{22}) \otimes C_{31} \otimes C_{32} \leq C_1 \otimes C_2 \otimes C_3 \leq C$, where the first refinement follows from (4.4). To summarize, sub-associativity is sufficient to guarantee independent implementability. \[Q.E.D.\]

**Comment 4.4** (counter-example to associativity). The following is a counter-example showing that the composition of contracts is not associative in general. Let there be the following components: $M_1, M_2, M_3, M_4, M_{12} = M_1 \times M_2, M_{23} = M_2 \times M_3, M_{123} = M_1 \times M_2 \times M_3$ (well defined, since $\times$ is associative), $M_{43} = M_4 \times M_3$, and $E$. Let the class $\mathcal{C}$ of contracts be composed of only the following contracts: (all assumptions are equal to the singleton $\{E\}$, so we will identify contracts with the set of their implementations): $\mathcal{C}_1 = \{M_1\}, \mathcal{C}_2 = \{M_2\}, \mathcal{C}_3 = \{M_3\}, \mathcal{C}_{12} = \{M_{12}, M_4\}, \mathcal{C}_{23} = \{M_{23}\}, \mathcal{C}_{(12)3} = \{M_{123}, M_{43}\}, \mathcal{C}_{(123)} = \{M_{123}\}$. Notice that in this class $\mathcal{C}$ of contracts, the contract $\mathcal{C}' = \{M_{12}\}$ does not exist. Now we compute the compositions:
4.4. Parallel composition

- $C_1 \otimes C_2 = C_{12}$, because $C_{12}$ is the only contract that satisfies the composition property, so it is also the minimum, even if it has the "spurious" component $M_4$;
- $C_2 \otimes C_3 = C_{23}$, because $C_{23}$ is the only contract that satisfies the composition property, so it is also the minimum, and has no spurious component;
- $(C_1 \otimes C_2) \otimes C_3 = C_{(12)3}$, (notice that $M_4$ actually composes with $M_3$);
- $C_1 \otimes (C_2 \otimes C_3) = C_{1(23)}$, (notice that $M_{43}$ here disappears, since neither $C_1$ nor $C_2 \otimes C_3$ have $M_4$ as an implementation).

This shows that contract composition is not associative in general. □

Property 6 (sub-distributivity). If the following contract compositions are all well defined, then the following holds:

$$[(C_{11} \land C_{21}) \otimes (C_{12} \land C_{22})] \leq [(C_{11} \otimes C_{12}) \land (C_{21} \otimes C_{22})] \quad (4.9)$$

Proof. By Lemma 4.2, $C_{(C_{11} \land C_{21}) \otimes (C_{12} \land C_{22})} \supseteq C_{C_{11} \otimes C_{12}}$. Taking the GLB of these two sets yields $[(C_{11} \land C_{21}) \otimes (C_{12} \land C_{22})] \leq C_{11} \otimes C_{12}$ and similarly for $C_{21} \otimes C_{22}$. Thus, (4.9) follows. □

Comment 4.5 (practical meaning of sub-distributivity). The practical meaning of sub-distributivity is best illustrated in the following context. Suppose the system under design decomposes into two subsystems labeled 1 and 2 and each subsystem has two viewpoints, labeled by another index with values 1 or 2 in such a way that contract $C_{11} \land C_{21}$ is the contract associated with subsystem 1 and $C_{12} \land C_{22}$ is the contract associated with subsystem 2. Thus, the left hand side of (4.9) specifies the set of implementations obtained by, first, implementing each subsystem independently, and then, composing these implementations. Property 6 states that, by doing so, we obtain an implementation of the overall contract obtained by, first, getting the two global viewpoints $C_{11} \otimes C_{12}$ and $C_{21} \otimes C_{22}$, and, then, taking their conjunction. This property supports independent implementation for specifications involving multiple viewpoints. Observe that only refinement, not equality, holds in (4.9). □
4.5 Quotient

The quotient of two contracts was defined in Tables 4.1 and 4.2:

\[
\mathcal{C}_1 / \mathcal{C}_2 = \max \{ \mathcal{C} \mid \mathcal{C} \otimes \mathcal{C}_2 \preceq \mathcal{C}_1 \} \quad (4.10)
\]

Its existence requires the following axiom:

**Axiom 3.** The “max” in (4.10) exists and is unique. □

This axiom can be equivalently reformulated as: the LUB \( \bigvee_{\mathcal{C}_2 \preceq \mathcal{C}_1} \mathcal{C} \) belongs to the set \( \{ \mathcal{C} \mid \mathcal{C} \otimes \mathcal{C}_2 \preceq \mathcal{C}_1 \} \); this LUB exists by Axiom 1. The quotient is the adjoint of the product operation \( \otimes \) in that \( \mathcal{C}_1 / \mathcal{C}_2 \) is the most general context \( \mathcal{C} \) in which \( \mathcal{C}_2 \) refines \( \mathcal{C}_1 \). It formalizes the practice of “patching” a component to make it behave according to another specification. From its definition (4.10), we deduce the following property:

**Property 7 (quotient).** We have \( \mathcal{C} \preceq \mathcal{C}_1 / \mathcal{C}_2 \iff \mathcal{C} \otimes \mathcal{C}_2 \preceq \mathcal{C}_1 \). □

4.6 Making contract composition associative

So far we only proved sub-associativity of our contract composition. In this section we introduce extra axioms that will indeed ensure associativity of the composition of contracts. In attempting to derive associativity, the missing step is the argument allowing us, in (4.6), to restrict the \( M_{12} \) to the special form \( M_{12} = M_1 \times M_2 \) for \( M_i \) ranging over the set of all implementations of \( \mathcal{C}_i \) for \( i = 1, 2 \). This is illegitimate in general, so we turn this into an extra axiom:

**Axiom 4.** Let \( \mathcal{C}_1, \mathcal{C}_2, \mathcal{C}' \) be three arbitrary contracts and \( M \in \mathcal{M} \) be any component such that \( M_1 \times M_2 \times M \) is well defined for any two components \( M_1 \) and \( M_2 \) implementing \( \mathcal{C}_1 \) and \( \mathcal{C}_2 \). We assume that the following properties (i) and (ii) are equivalent:

\[
\begin{align*}
\forall M_{12} &\vdash^u \mathcal{C}_1 \otimes \mathcal{C}_2 \quad \implies \quad M_{12} \times M \vdash^{u*} \mathcal{C}' & (i) \\
\forall M_i &\vdash^u \mathcal{C}_i, i = 1, 2 \quad \implies \quad M_1 \times M_2 \times M \vdash^{u*} \mathcal{C}' & (ii)
\end{align*}
\]

where \( \vdash^{u*} \) denotes either \( \vdash^u \) on both properties or \( \vdash^e \) on both properties. □

**Theorem 4.6.** Under extra Axiom 4, contract composition is associative. □
Proof. It suffices to show that, in property (4.6) it is legitimate to restrict \( M_{12} \) to the special form \( M_1 \times M_2 \). But this is a direct consequence of Axiom 4 applied for \( |=^u \) and \( M = M_3 \), and for \( |=^c \) and \( M = E \). □

The counterexample to associativity that was presented in Comment 4.4 violates Axiom 4, since, in the considered class \( C \) of contracts defined for that example, the contract \( \mathcal{C}' = \{ M_{12} \} \) does not exist.

The following easy result gives a sufficient condition for Axiom 4 to hold. Recall that \( C \) denotes the underlying class of contracts:

**Theorem 4.7.** The following condition ensures that \( C \) satisfies Axiom 4: the underlying class \( \mathcal{M} \) of all components is equipped with a partial order \( \sqsubseteq \) such that

1. For every \( M_1, M'_1, M_2 \) such that \( M'_1 \sqsubseteq M_1 \), we have \( (M'_1 \times M_2) \sqsubseteq (M_1 \times M_2) \);

2. For every compatible \( \mathcal{C} \in \mathcal{C}, \mathcal{E}_{\mathcal{C}} \) is \( \sqsubseteq \)-closed and possesses a unique \( \sqsubseteq \)-maximal environment \( \mathcal{E}_{\mathcal{C}} \);

3. For every consistent \( \mathcal{C} \in \mathcal{C}, \mathcal{M}_{\mathcal{C}} \) possesses a unique \( \sqsubseteq \)-maximal component \( m_{\mathcal{C}} \); furthermore, every \( M \in \mathcal{M} \) satisfying \( m_{\mathcal{C}} \sqsubseteq M \sqsubseteq M_{\mathcal{C}} \) is an implementation of \( \mathcal{C} \);

4. For every consistent composition \( \mathcal{C}_1 \otimes \mathcal{C}_2, \mathcal{M}_{\mathcal{C}_1 \otimes \mathcal{C}_2} = \mathcal{M}_{\mathcal{C}_1} \times \mathcal{M}_{\mathcal{C}_2} \) and \( m_{\mathcal{C}_1 \otimes \mathcal{C}_2} = m_{\mathcal{C}_1} \times m_{\mathcal{C}_2} \) both hold. □

As we shall see later, extra Axiom 4 holds for Assume/Guarantee contracts but fails to hold for several Interface frameworks, see Section 8.3.

### 4.7 Abstractions

An abstraction consists of an abstract domain of contracts — intended to be simple enough to support analysis — together with a mapping, from contracts (we call them “concrete contracts” in the sequel) to abstract contracts. The hope is that properties of contracts can be proved by taking abstractions thereof.

In this section we explain how to lift, to contracts, abstraction procedures available on components. In doing so, our objectives are the following:
1. Abstraction for contracts should allow proving refinement, consistency, or compatibility, for any contract or sets of contracts, based on their abstractions;

2. Properties of contracts should be deducible from their abstractions, compositionally with respect to both conjunction and parallel composition;

3. The mechanism of lifting abstractions, from components to contracts should be generic and instantiable for any concrete contract framework.

A large part of this agenda has been achieved, as we shall see. Our starting point is a framework for abstracting components. This framework must be rich enough to support abstraction and its opposite operation in a coherent way. A known formalization of this is the notion of Galois connection, which is key in the theory of Abstract Interpretation [92, 93, 94, 202]. A Galois connection consists of two concrete and abstract partially ordered sets \((X_c, \sqsubseteq_c)\) and \((X_a, \sqsubseteq_a)\), and two total monotonic maps:

\[
\alpha : X_c \mapsto X_a : \text{the abstraction}
\]

\[
\gamma : X_a \mapsto X_c : \text{the concretization}
\]

such that, for any two \(X_c \in X_c\) and \(X_a \in X_a\),

\[
X_c \sqsubseteq_c \gamma(X_a) \quad \text{if and only if} \quad \alpha(X_c) \sqsubseteq_a X_a \tag{4.12}
\]

Property (4.12) is equivalent to any of the following properties:

\[
X_c \sqsubseteq_c \gamma \circ \alpha(X_c) \quad ; \quad \alpha \circ \gamma(X_a) \sqsubseteq_a X_a \tag{4.13}
\]

where \(\gamma \circ \alpha\) is the composition of the two referred maps: \(\gamma \circ \alpha(X_c) \overset{\text{def}}{=} \gamma(\alpha(X_c))\). The intent is that \(X_c\) is the concrete domain of interest and \(X_a\) is a simpler and coarser representation of the former, where concrete entities can be approximated. The two orders \(\sqsubseteq_c/\sqsubseteq_a\) are interpreted as “is more precise” — for example, if components are specified as sets of behaviors, the preciseness order is simply set inclusion. The Galois connection property (4.12) relates the preciseness orders in concrete and abstract domains.

\[f : X \mapsto Y, \text{ where } (X, \leq_X) \text{ and } (Y, \leq_Y) \text{ are two ordered sets, is monotonic if } x' \leq_X x \text{ implies } f(x') \leq_Y f(x), \text{ and strictly monotonic if } x' <_X x \text{ implies } f(x') <_Y f(x), \text{ where } < \overset{\text{def}}{=} \leq \cap \neq.\]
4.7. Abstractions

Having the above notions at hand, our next step consists in systematically lifting a given Galois connection \((\alpha, \gamma)\) on components to an abstraction on contracts. Since contracts are defined as pairs consisting of a set of valid environments and a set of valid implementations, our first task is to lift Galois connections on sets, to abstractions on powersets. Our construction will be using the notion of inverse map, which we recall next. For \(X\) and \(Y\) two sets, \(f : X \to Y\) a partial function, and \(Z \subseteq Y\), define

\[
f^{-1}(Z) = \{ x \in X \mid f(x) \text{ is defined and } f(x) \in Z \}
\]

The following holds:

\[
\begin{align*}
  f^{-1}(Z_1 \cap Z_2) &= f^{-1}(Z_1) \cap f^{-1}(Z_2) \\
  f^{-1}(Z_1 \cup Z_2) &= f^{-1}(Z_1) \cup f^{-1}(Z_2)
\end{align*}
\]

(4.14)

Referring to the previously introduced notations, we consider the sets \(X^c \subseteq \wp(X^c)\) and \(X^a \subseteq \wp(X^a)\) collecting all ideals\(^2\) of \((X^c, \sqsubseteq^c)\) and \((X^a, \sqsubseteq^a)\), respectively. Equip \(X^c\) and \(X^a\) with their inclusion orders \(\subseteq^c\) and \(\subseteq^a\). The canonical abstraction

\[
\overline{\alpha} : (X^c, \subseteq^c) \to (X^a, \subseteq^a)
\]

associated to Galois connection \((\alpha, \gamma)\) is defined by

\[
\overline{\alpha}(\chi^c) = \gamma^{-1}(\chi^c)
\]

(4.15)

where \(\chi^c\) ranges over \(X^c\). Definition (4.15) is sound since \(\gamma\) is monotonic. The following property follows by construction:

\[
\forall \chi^c \in X^c: \chi^c = \emptyset \implies \overline{\alpha}(\chi^c) = \emptyset
\]

(4.16)

We now instantiate the generic construction (4.15) by substituting \(X^c \leftarrow M^c\) and \(X^a \leftarrow M^a\), where \(M^c\) and \(M^a\) are concrete and abstract domains of components. We assume the following, which expresses that the preciseness orders fit our contract framework:

**Axiom 5.** For any concrete contract \(\mathcal{C}^c \in \mathcal{C}_c\) with semantics \([\mathcal{C}^c] = (E_{\mathcal{C}^c}, M_{\mathcal{C}^c})\), both \(E_{\mathcal{C}^c}\) and \(M_{\mathcal{C}^c}\) are downward closed under \(\subseteq^c\). The same holds for abstract contracts.

\(^2\)An ideal of \((X, \subseteq)\) is a \(\subseteq\)-downward closed subset of \(X\).
A Mathematical Meta-Theory of Contracts

Axiom 5 is indeed very natural for known contract frameworks, see Section 5.3 and [45] for more details. By (4.15) we inherit an abstraction \( \overline{\alpha} \) from \( (\mathcal{M}_c^c, \subseteq) \) to \( (\mathcal{M}_a^a, \subseteq) \). Since the semantics of a concrete generic contract \( C_c \) is
\[
\llbracket C_c \rrbracket = (E_c, M_c) \in \mathcal{M}_c^c \times \mathcal{M}_c^c,
\]
we can define the abstraction \( \overline{\alpha}(C_c) \) of \( C_c \), whose semantics is:
\[
\llbracket \overline{\alpha}(C_c) \rrbracket = \text{def} \ (\overline{\alpha}(E_c), \overline{\alpha}(M_c)) \in \mathcal{M}_a^a \times \mathcal{M}_a^a \quad (4.17)
\]
\( \overline{\alpha} \) defined by (4.17) is the canonical abstraction on contracts associated to the Galois connection \((\alpha, \gamma)\) on components. The following theorem achieves our first objectives regarding contract abstraction:

**Theorem 4.8.** For \( M_c, E_c, C_c \) any concrete component, environment, and contract:
\[
\begin{align*}
\alpha(M_c) &\models_a \overline{\alpha}(C_c) \implies M_c \models_c C_c \\
\alpha(E_c) &\models_a \overline{\alpha}(C_c) \implies E_c \models_c C_c \quad (4.18) \\
C_c' \preceq_c C_c &\implies \overline{\alpha}(C_c') \preceq_a \overline{\alpha}(C_c) \quad (4.19) \\
\overline{\alpha}(C_c) \text{ compatible} &\implies C_c \text{ compatible} \quad \text{resp. consistent} \implies C_c \text{ compatible} \quad \text{resp. consistent} \quad (4.20)
\end{align*}
\]

**Proof.** Property (4.20) follows immediately from (4.16). Focus next on (4.19). Since set abstraction \( \overline{\alpha} \) is monotonic with respect to set inclusion, we deduce that contract abstraction \( \overline{\alpha} \) is monotonic for the pair of refinement orders \( \leq_c, \leq_a \). Regarding (4.18), \( \alpha(M_c) \models_a \overline{\alpha}(C_c) \) means that \( \gamma(\alpha(M_c)) \in \mathcal{M}_{\overline{\alpha}c} \). By (4.13), \( M_c \subseteq_c \gamma(\alpha(M_c)) \), which, by Axiom 5, implies \( M_c \in \mathcal{M}_{\overline{\alpha}c} \), i.e., \( M_c \models_c C_c \). Similarly, \( \alpha(E_c) \models_a \overline{\alpha}(C_c) \) means that \( \gamma(\alpha(E_c)) \in \mathcal{E}_{\overline{\alpha}c} \). By (4.13), \( E_c \subseteq_c \gamma(\alpha(E_c)) \), which, by Axiom 5, implies \( E_c \in \mathcal{E}_{\overline{\alpha}c} \), i.e., \( E_c \models_c C_c \). \( \square \)

Observe that (4.18) allows proving implementation and environment relations based on abstractions. Similarly, (4.20) allows proving compatibility or consistency based on abstractions. In contrast, (4.19) allows disproving refinement based on abstractions.

The second part of our agenda is about the compositionality of the abstraction, with respect to both the conjunction and the parallel composition. Observe first that Property (4.19) of Theorem 4.8 implies the refinement \( \overline{\alpha}(C_c^1 \wedge C_c^2) \preceq_a \overline{\alpha}(C_c^1) \wedge \overline{\alpha}(C_c^2) \). Using, however, the fact that the abstraction and the concretization for powersets arise from inverse maps, we can in fact get equalities:
4.7. Abstractions

Theorem 4.9. The following equalities hold:
\[
\overline{\alpha}(\mathcal{C}_1 \land \mathcal{C}_2) = \overline{\alpha}(\mathcal{C}_1^\gamma) \land \overline{\alpha}(\mathcal{C}_2^\gamma)
\] (4.21)

Proof. The set of implementations of the GLB \(\mathcal{C}_1 \land \mathcal{C}_2\) is given by
\[
\mathcal{M}_{\mathcal{C}_1 \land \mathcal{C}_2} = \max\left\{ \mathcal{M}_c \mid \mathcal{M}_c \in \mathbb{M}_c^\gamma \text{ and } \mathcal{M}_c \subseteq \mathcal{M}_{\mathcal{C}_1^\gamma} \cap \mathcal{M}_{\mathcal{C}_2^\gamma} \right\}
\]
Consequently, the set of implementations of the abstraction of this GLB is given by
\[
\mathcal{M}_{\overline{\alpha}(\mathcal{C}_1 \land \mathcal{C}_2)} = \gamma^{-1}\left(\mathcal{M}_{\mathcal{C}_1 \land \mathcal{C}_2}\right)
\] (4.22)
\[
= \max\left\{ \mathcal{M}_a \mid \mathcal{M}_a \in \mathbb{M}_a^\gamma \text{ and } \gamma(\mathcal{M}_a) \subseteq \mathcal{M}_{\mathcal{C}_1^\gamma} \cap \mathcal{M}_{\mathcal{C}_2^\gamma} \right\}
\]
(by (4.14)) = \max\left\{ \mathcal{M}_a \cap \gamma^{-1}\left(\mathcal{M}_{\mathcal{C}_1^\gamma} \cap \mathcal{M}_{\mathcal{C}_2^\gamma}\right) \right\}
\[
= \max\left\{ \mathcal{M}_a \mid \mathcal{M}_a \in \mathbb{M}_a^\gamma \text{ and } \mathcal{M}_a \subseteq \mathcal{M}_{\overline{\alpha}(\mathcal{C}_1^\gamma)} \text{ and } \mathcal{M}_a \subseteq \mathcal{M}_{\overline{\alpha}(\mathcal{C}_2^\gamma)} \right\}
\]
Similarly,
\[
\mathcal{E}_{\overline{\alpha}(\mathcal{C}_1 \land \mathcal{C}_2)} = \min\left\{ \mathcal{E}_a \mid \mathcal{E}_a \in \mathbb{E}_a^\gamma \text{ and } \mathcal{E}_a \geq \mathcal{E}_{\overline{\alpha}(\mathcal{C}_1^\gamma)} \right\}
\] (4.23)

(4.22) and (4.23) together prove the theorem.

We now focus on the parallel composition of contracts. We wish to relate \(\overline{\alpha}(\mathcal{C}_1^\gamma) \otimes \overline{\alpha}(\mathcal{C}_2^\gamma)\) and \(\overline{\alpha}(\mathcal{C}_1^\gamma \otimes \mathcal{C}_2^\gamma)\). Unlike previous properties, this does not come for free. We first need an additional property for the concretization of components: \(\gamma\) is called \textit{sub-multiplicative} if
\[
\gamma(X_a \times_a X_a) \sqsubseteq_c \gamma(X_a^1) \times_c \gamma(X_a^2)
\] (4.24)
and \textit{multiplicative} if equality holds in (4.24). We finally assume the following:

\textbf{Axiom 6.} Extra Axiom 4 is in force for both concrete and abstract contracts and the concretization map \(\gamma : \mathbb{M}_a \mapsto \mathbb{M}_c\) satisfies the following property:

\(\forall \mathcal{C}_{a,1}, \mathcal{C}_{a,2}, \mathcal{C}_c\) any three contracts, the following two properties are equivalent:
\[
\forall \mathcal{M}_1 \models_a \mathcal{C}_{a,1} \otimes \mathcal{C}_{a,2} \implies \gamma(\mathcal{M}_1) \models_c \mathcal{C}_c
\]
\[
\forall \mathcal{M}_i \models_a \mathcal{C}_{a,i}, i = 1, 2 \implies \gamma(\mathcal{M}_1 \times \mathcal{M}_2) \models_c \mathcal{C}_c'
\]
Theorem 4.10. Extra Axioms 4–6 are in force. If \( \gamma \) is sub-multiplicative, then
\[
\overline{\alpha}(\mathcal{C}_c^1) \otimes \overline{\alpha}(\mathcal{C}_c^2) \leq_a \overline{\alpha}(\mathcal{C}_c^1 \otimes \mathcal{C}_c^2) \quad (4.25)
\]

Proof. Set \( \mathcal{C}_c = \mathcal{C}_c^1 \otimes \mathcal{C}_c^2 \) and \( \mathcal{C}_a = \overline{\alpha}(\mathcal{C}_c^1) \otimes \overline{\alpha}(\mathcal{C}_c^2) \).

Focus on implementations: We need to prove that \( M_{\mathcal{C}_a} \subseteq M_{\overline{\alpha}(\mathcal{C}_c)} \), i.e., using (4.17): \( M_{\mathcal{C}_a} \subseteq \overline{\alpha}(M_{\mathcal{C}_c}) \), which is, by (4.15), equivalent to
\[
\forall M_a : M_a \models_a \mathcal{C}_a \implies \gamma(M_a) \models_\mathcal{C}_c \quad (4.26)
\]
By Axiom 6 applied to \( \gamma \) it is enough to prove (4.26) when \( M_a \) has the restricted form \( M_a = M_a^1 \times_a M_a^2 \), where \( M_a^i \models_a \overline{\alpha}(\mathcal{C}_c^i) \), which, by using (4.17), is equivalent to \( \gamma(M_a^i) \models_\mathcal{C}_c^i \). Thus (4.26) amounts to proving, for any two abstract components \( M_a^1, M_a^2 \):
\[
\gamma(M_a^i) \models_\mathcal{C}_c^i \implies \gamma(M_a^1 \times_a M_a^2) \models_\mathcal{C}_c \quad (4.27)
\]
By definition of the contract composition, \( \gamma(M_a^i) \models_\mathcal{C}_c^i \) implies \( \gamma(M_a^1 \times_a \gamma(M_a^2)) \models_\mathcal{C}_c \gamma(\mathcal{C}_c^1 \otimes \mathcal{C}_c^2) = \mathcal{C}_c \). Since \( \gamma \) is sub-multiplicative, we have \( \gamma(M_a^1 \times_a M_a^2) \subseteq_\mathcal{C} \gamma(M_a^1) \times_\mathcal{C} \gamma(M_a^2) \) and we deduce (4.27) by invoking Axiom 5. This proves the implementation part of (4.25).

Focus next on environments: We need to prove that \( E_{\mathcal{C}_a} \supseteq E_{(\overline{\alpha}(\mathcal{C}_c))} \), i.e., using (4.17): \( E_{\mathcal{C}_a} \supseteq \overline{\alpha}(E_{\mathcal{C}_c}) \), which is, by (4.15), equivalent to
\[
\forall E_a : E_a \models_a \mathcal{C}_a \iff \gamma(E_a) \models_\mathcal{C}_c \quad (4.28)
\]
By definition of the contract composition, the left hand side of (4.28) is equivalent to: for any \( M_a^i \models_\mathcal{C}_c^i \), the following holds: \( E_a \times_a M_a^i \models_a \overline{\alpha}(\mathcal{C}_c^i) \), where \( j \neq i \), which is equivalent to \( \gamma(E_a \times_a M_a^i) \models_\mathcal{C}_c^i \). On the other hand, the right hand side of (4.28) is equivalent to: for any \( M_a^i \models_\mathcal{C}_c^i \), the following holds: \( \gamma(E_a) \times_\mathcal{C} M_a^i \models_\mathcal{C}_c^i \) where \( j \neq i \). To summarize it is enough to prove the following, for any abstract environment \( E_a \):
\[
\begin{align*}
\forall M_a^i : M_a^i \models_\mathcal{C}_c^i &\implies \downarrow \quad (4.29) \\
\gamma(E_a) \times_\mathcal{C} M_a^i \models_\mathcal{C}_c^i &\implies \downarrow
\end{align*}
\]
and it even suffices to prove (4.29) when we restrict the quantification in the left hand side of (4.29) to the subset of $M'_c$ of the form $M'_c = \gamma(M'_a)$. Since $\gamma$ is sub-multiplicative, we get $\gamma(E_a \times_a M'_a) \sqsubseteq \gamma(E_a) \times_c \gamma(M'_a)$, which, by Axiom 5, implies the right hand side of (4.29) and proves the theorem. □

Concluding discussion regarding contract abstraction

- From having a Galois connection on components we inherit an abstraction on contracts that is monotonic with respect to the refinement orders. Consistency and compatibility can both be checked on abstractions, see Theorem 4.8. The reader may conjecture that it should be possible to construct a Galois connection for contracts. We are rather convinced that this is not achievable, see [45] regarding obstructions.

- Theorem 4.9 allows proving consistency and compatibility in a $\land$-modular way by using the equality $\bar{\alpha}(\bigwedge_{i \in I} \mathcal{C}_i) = \bigwedge_{i \in I} \bar{\alpha}(\mathcal{C}_i)$.

- If $\gamma$ is sub-multiplicative, Theorem 4.10 allows proving consistency in a $\otimes$-modular way by using refinement $\bigotimes_{i \in I} \bar{\alpha}(\mathcal{C}_i) \preceq_a \bar{\alpha}(\bigotimes_{i \in I} \mathcal{C}_i)$. This inequality is in the wrong way, however, for proving compatibility in a $\otimes$-modular way. Regarding this theorem, Galois connections on components where concretization is sub-multiplicative are quite natural. Thus, Theorem 4.10 will be easy to have. See [45] for variations of Theorem 4.10.

4.8 Bibliographical note on abstract contract theories

Our presentation here is new. The important notion of sub-associativity is new. Then, as far as we know, no notion of abstraction existed for contracts or specifications, with the attempt of being compliant with contract relations and operators. Our proposal here is new.

There is only a small literature providing an abstract formalization of the notion of contracts. The only attempts we are aware of are the recent works by Bauer et al. [29] and Chen et al. [85], albeit with deeply different and complementary approaches.

The publication [29] develops an axiomatization of the notion of specification, from which contracts can be derived in a second step. More precisely,
specifications are abstract entities that obey the following list of axioms: it possesses a refinement relation that is a preorder, which induces a notion of equivalence of specifications, and a parallel composition that is associative, commutative (modulo equivalence), and monotonic with respect to refinement. It is assumed that, if two specifications possess a common lower bound, then they possess a greatest lower bound. A quotient is also assumed, which is the residuation of the parallel composition. From specifications, contracts are introduced as pairs of specifications, very much like Assume/Guarantee contracts we develop in Chapter 5 are pairs of assertions. Sets of valid environments and sets of implementations are associated to contracts. Finally, modal contracts are defined by borrowing ideas from modal specifications that we discuss in Chapter 8. This abstract theory nicely complements the one we develop here in that it shows that both specifications and contracts can be defined as primitive entities and be used to build more concrete theories.

The work [85] develops the concept of declarative specification, which consists of a tuple \( \mathcal{P} = (\Sigma_{\text{in}}, \Sigma_{\text{out}}, T_{\Sigma}, F_{\Sigma}) \), where \( \Sigma_{\text{in}} \) and \( \Sigma_{\text{out}} \) are input and output alphabets of actions, \( \Sigma = \Sigma_{\text{in}} \uplus \Sigma_{\text{out}} \), and \( T_{\Sigma}, F_{\Sigma} \subseteq \Sigma^* \) such that \( F_{\Sigma} \subseteq T_{\Sigma} \) are sets of permissible and inconsistent traces, respectively — this approach find its origins in earlier work by Dill [116] and Negulescu [205]. Outputs are under the control of the component, whereas inputs are issued by the environment. Thus, after any successful interaction between the component and the environment, the environment can issue any input \( \alpha \), even if it will be refused by the component. If \( \alpha \) is refused by the component after the trace \( t \in T_{\Sigma}, t.\alpha \in F_{\Sigma} \) is an inconsistent trace, capturing that a communication mismatch has occurred. An environment is called safe if it can prevent a component from performing an inconsistent trace. For \( Q \) to be used in place of \( \mathcal{P} \) it is required that \( Q \) must exist safely in any environment that \( \mathcal{P} \) can exist in safely; this is the basis on which refinement is defined. Alphabet extension is used, by which input actions outside the considered alphabet are followed by an arbitrary behavior for the declarative specification. A conjunction is proposed that is the GLB for refinement order. A parallel composition is proposed, which is monotonic with respect to refinement. A quotient is also proposed, which is the residuation of parallel composition. In the same direction, an algebraic theory of interface automata is proposed in [86], paying special attention to
4.8. Bibliographical note on abstract contract theories

issues of safety (which is usual) and progress (which is not usual). Finally, [229] proposes a mathematical basis for multi-view modeling and [213] was an early paper proposing a notion of quotient for an interface model.
Our static example of Section 2 provided an example of contract specified using Assumptions and Guarantees. In Assume/Guarantee contracts (A/G contracts), Assumptions characterize the valid environments for the considered component, whereas the Guarantees specify the commitments of the component itself, when put in interaction with a valid environment. Various kinds of A/G contract theories can be obtained by specializing the meta-theory in different ways. Variations concern how the composition of components is defined. We will review some of these specializations and relate them to the existing literature.

In general, A/G contract theories build on top of component models that are assertions, i.e., sets of behaviors or traces assigning successive values to variables. As we shall see, different kinds of frameworks for assertions can be considered, including asynchronous frameworks of Kahn Process Networks (KPN) [167] and synchronous frameworks in which behaviors are sequences of successive reactions assigning values to the set of variables of the considered system.

We first develop the theory for the simplest case of a fixed alphabet of variables and explain how it specializes to the Moore Interfaces proposed by [82]. Then, we develop the other cases.
5.1 Synchronous A/G contracts with fixed alphabet

For this simplest variant, all components and contracts involve the same finite alphabet $V$ of variables possessing identical domain $D$. The restriction to a fixed alphabet of variables in this section is intended to simplify the concepts and the formulas. It is by no means essential. We will deal with variable alphabets later, in Section 5.2.

Synchronous assertions, which constitute the basis of synchronous A/G-components and contracts, are introduced next. A reaction assigns to each variable of $V$ a value from its domain: $s \in D^V$. By adding a distinguished symbol $\bot \not\in D$ to model the absence of an actual variable in the considered reaction, we get the multiple-clocked synchronous model used by synchronous languages [44]. Denote by $\varepsilon = \bot^V$ the silent reaction, assigning $\bot$ to every variable. A synchronous behavior $\sigma$ is a finite or infinite sequence of reactions. A synchronous assertion $P$ is a set of synchronous behaviors:

$$P \subseteq (V \mapsto D \cup \{\bot\})^\omega.$$  (5.1)

Synchronous assertions are equipped with the set algebra $\cap$, $\cup$, $\neg$, where $\neg$ denotes set complement. From now on and unless otherwise mentioned, we omit the term “synchronous”. We discuss in Section 5.5 variants of this framework with asynchronous models of behaviors.

**Definition 5.1 ([41]).** Say that assertion $P$ is stuttering invariant if:

1. $P$ is closed under the transformations

$$\sigma = s_1, \ldots, s_k, s_{k+1}, \ldots \mapsto \text{stretch}_k(\sigma) = s_1, \ldots, s_k, \varepsilon, s_{k+1}, \ldots$$  (5.2)

where $k$ is an arbitrary integer — inserting at any time $k$ a silent reaction in a behavior of $P$ still yields a behavior of $P$. and

2. $P$ is a closed set when $(V \mapsto (D \cup \{\bot\}))^\omega$ is equipped with the product discrete topology.

In particular, if $P$ is stuttering invariant, then by using condition 1 of stuttering invariance, it contains behaviors beginning with the silent behavior $\varepsilon^k$ with an arbitrary length $k$. By condition 2 of stuttering invariance, the behavior $\varepsilon^\omega$ having only silent reactions, which is the limit with respect to
Assume/Guarantee Contracts

the product topology of a sequence of behaviors beginning by \( \varepsilon^k \), also belongs to \( P \). Stuttering invariance is a desirable property for an open system, since it may be subsequently put in an environment that is acting when the considered system is sleeping.

**Definition 5.2.** A component is any stuttering invariant assertion. □

Thus, it is always allowed for a component to do nothing. The class of components is stable under intersection. Two components are always composable and we define component composition by the intersection of their respective assertions:

\[
P_1 \times P_2 = P_1 \cap P_2
\]  

Formulas (5.1) and (5.3) define a framework of synchronous components. It coincides with the framework used in [41].

Despite the fact that a component is typically implemented in practice in the form of a program, we intentionally define it in Definition 5.2 as an abstract assertion. This definition gives us greater flexibility and does not enforce any particular syntax. The abstract assertions are, however, not effective and their syntax must be fixed in order to allow finite description of the component behavior. The choice of the syntax is crucial and affects both the expressiveness and succinctness of the assertion language.¹

**Definition 5.3.** A contract is a pair \( \mathcal{C} = (A, G) \) of assertions, called the assumptions and the guarantees. The set \( E_\mathcal{C} \) of the legal environments for \( \mathcal{C} \) collects all components \( E \) such that \( E \subseteq A \). The set \( M_\mathcal{C} \) of all components implementing \( \mathcal{C} \) is defined by \( A \times M \subseteq G \). □

Observe that we are not requiring any particular condition on the sets \( A \) and \( G \). In particular, they may not contain \( \varepsilon \) and may even be empty. For this section, the underlying set \( \mathcal{C} \) of contracts is the set of all pairs \( (A, G) \) of assumptions and guarantees as defined above. By Definition 5.2,

\[
\text{contract } \mathcal{C} = (A, G) \text{ is compatible if and only if } \varepsilon^\omega \in A, \\
\text{and in this case } E_\mathcal{C} = A \text{ is the maximal (for set inclusion) environment of } \mathcal{C}.
\]  

¹Scade and the discrete time part of Simulink or Modelica are examples of concrete languages to specify components in a synchronous data flow style, through input-output maps. The graphs of these maps are the assertions in our sense.
Denoting by $\neg A$ the complement of set $A$, any component $M$ such that $M \subseteq G \cup \neg A$ is an implementation of $\mathcal{C}$. Thus, contract $\mathcal{C} = (A, G)$ is consistent if and only $\varepsilon^\omega \in G \cup \neg A$, and in this case $M_{\mathcal{C}} = G \cup \neg A$ is the maximal (for set inclusion) implementation of $\mathcal{C}$. Thus, contract $\mathcal{C} = (A, G)$ is consistent if and only $\varepsilon^\omega \in G \cup \neg A$, and in this case $M_{\mathcal{C}} = G \cup \neg A$ is the maximal (for set inclusion) implementation of $\mathcal{C}$.

Observe that two contracts $\mathcal{C}$ and $\mathcal{C}'$ with identical alphabets of variables, identical assumptions $A' = A$, and such that $G' \cup \neg A' = G \cup \neg A$, possess identical sets of implementations: $M_{\mathcal{C}'} = M_{\mathcal{C}}$. According to our meta-theory, such two contracts are equivalent. Say that contract $\mathcal{C} = (A, G)$ is saturated if $G = G \cup \neg A$, or, equivalently, if $G \cup A = \Omega$, (5.6)

where $\Omega \overset{\text{def}}{=} (V \mapsto D \cup \{\bot\})^\omega$ is the trivial assertion collecting all behaviors. Contract $\mathcal{C} = (A, G)$ is equivalent to its saturated form $(A, G \cup \neg A)$. Refinement, conjunction, and parallel composition are defined as follows, for $A/G$ contracts in saturated form:

**Definition 5.4.** Let $\mathcal{C}_1$ and $\mathcal{C}_2$ be two saturated contracts with identical alphabets of variables.

1. Say that $\mathcal{C}_2$ refines $\mathcal{C}_1$, written $\mathcal{C}_2 \preceq \mathcal{C}_1$, iff $A_2 \supseteq A_1$ and $G_2 \subseteq G_1$;

2. The conjunction of $\mathcal{C}_1$ and $\mathcal{C}_2$ is defined as being the corresponding GLB: $\mathcal{C}_1 \land \mathcal{C}_2 \overset{\text{def}}{=} (A_1 \cup A_2, G_1 \cap G_2)$;

3. The parallel composition of $\mathcal{C}_1$ and $\mathcal{C}_2$, denoted by $\mathcal{C}_1 \otimes \mathcal{C}_2$, is defined as being the pair $(A, G)$ such that

$$G = G_1 \cap G_2$$

and

$$A = (A_1 \cap A_2) \cup (G_1 \cap G_2)$$

(5.7)

**Summary of Results 1.** Definitions 5.2–5.4 instantiate the meta-theory. Furthermore, extra Axiom 4 of the meta-theory holds, thus ensuring that contract composition is associative.

**Comment 5.1** (practical meaning of contract conjunction). It is worth discussing the practical meaning of contract conjunction as defined above, for non-saturated contracts. Consider two non-saturated contracts $\mathcal{C}_1$ and $\mathcal{C}_2$. 
Their conjunction is $C_1 \land C_2 = (A_1 \cup A_2, (G_1 \cup \neg A_1) \cap (G_2 \cup \neg A_2))$. What happens if we subject this contract to an environment $E$ that is legal for $C_1 \land C_2$ but satisfies $E \subseteq A_1$ and $E \not\subseteq A_2$? Let $M$ be an implementation of the conjunction $C_1 \land C_2$. Then we have by definition $E \times M \subseteq (G_1 \cup \neg A_1) \cap (G_2 \cup \neg A_2)$ which, since $E \subseteq A_1$, boils down to $E \times M \subseteq G_1 \cap (G_2 \cup \neg A_2)$. This formula expresses that $G_1$ is guaranteed in this case, but $G_2$ is not, due to the extra term $\neg A_2$. Thus, formula (2) reflects the intuition that, in the conjunction $C_1 \land C_2$, if one assumption, e.g., $A_2$, fails to be met by an environment, the corresponding contract $C_2$ is relieved from its guarantee, but the other contract $C_1$ “survives”.

We now develop the theorems that support the above summary of results.

**Theorem 5.2.** With reference to Definition 5.4, for $C_1$ and $C_2$ two saturated A/G contracts:

1. $C_2 \preceq C_1$ holds if and only if $C_2$ refines $C_1$ in the sense of the meta-theory;
2. $C_2 \land C_1$ coincides with the conjunction of $C_1$ and $C_2$ in the sense of the meta-theory;
3. $C_2 \otimes C_1$ coincides with the composition of $C_1$ and $C_2$ in the sense of the meta-theory.

Furthermore, the Axioms 1 (regarding GLB only) and 2 of the meta-theory hold.

**Proof.** Statement 1 is immediate from Definition 5.3 and the definition (5.6) of contract saturation. Regarding statement 2, note that the so defined contract is again saturated since $(G_1 \cap G_2) \cup (A_1 \cup A_2) = (G_1 \cup A_1 \cup A_2) \cap (G_2 \cup A_1 \cup A_2) \supseteq (G_1 \cup A_1) \cap (G_2 \cup A_2) = \Omega$, since $G_i \cup A_i = \Omega$ for $i = 1, 2$. No corresponding formula exists for the Least Upper Bound $\lor$.

We next move to statement 3 regarding contract composition — unlike previous statements, this one is non trivial. We use the notation $C_{C_1 \land C_2}$ introduced in Section 4.4. In a first step we prove the following, where $C_{C_1 \land C_2}$ is defined in (4.2):

$$C_{C_1 \land C_2} = C_{1,2}$$ (5.8)
5.1. Synchronous A/G contracts with fixed alphabet

\[ C_{1,2} = \begin{cases} \text{(A', G') saturated} & \quad \text{and} \quad G' \supseteq G_1 \cap G_2 \text{ and } A' \cap G_2 \subseteq A_1 \\ \text{and} \quad A' \cap G_1 \subseteq A_2 \end{cases} \] (5.9)

Consider first the case where one of the two contracts, say, \( \mathcal{C}_1 \), is inconsistent, see (5.5). Since \( \mathcal{C}_1 \) is saturated, it follows that its assumption is trivially permissive. Then every saturated contract \( (A', G') \) satisfies the conditions defining \( C_{1,2} \). Inspecting (4.2) for the case of an inconsistent contract \( \mathcal{C}_1 \) with trivially permissive assumption shows that every contract \( \mathcal{C} \) belongs to it. Thus (5.8) holds in this case.

We can thus assume in the rest of the proof of (5.8) that both contracts \( \mathcal{C}_{i}, i = 1, 2 \) are consistent, i.e., \( G_i \neq \emptyset, i = 1, 2 \). We first prove the inclusion \( \mathcal{C}_{\mathcal{C}_1, \mathcal{C}_2} \supseteq \mathcal{C}_{1,2} \). Select \( (A', G') \in \mathcal{C}_{1,2} \). We distinguish two cases:

1. Case \( (A', G') \) is compatible. In this case we can pick an environment \( E \subseteq A' \). On the other hand, pick \( M_1 \subseteq G_1, M_2 \subseteq G_2 \) (this ensures that \( M_i \) implements \( \mathcal{C}_i \) since the pairs are saturated) and get, by using (5.3) and definition of \( C_{1,2} \): \( M_1 \times M_2 = M_1 \cap M_2 \subseteq G' \). In the same way, we derive \( E \times M_2 \subseteq E \cap G_2 \subseteq A' \cap G_2 \subseteq A_1 \) and, similarly, \( E \times M_1 \subseteq A_2 \). Thus, the triple \( (M_1, M_2, E) \) satisfies the three conditions on the right most column of (4.2). This shows that the considered pair \( (A', G') \) belongs to \( \mathcal{C}_{\mathcal{C}_1, \mathcal{C}_2} \).

2. Case \( (A', G') \) is incompatible. Since no environment exists for the contract \( (A', G') \) in this case, the last two conditions on the right most column of (4.2) are vacuously satisfied. The same argument as for the case 1 shows that the first condition on the right most column of (4.2) is satisfied. Thus, again, the considered pair \( (A', G') \) belongs to \( \mathcal{C}_{\mathcal{C}_1, \mathcal{C}_2} \).

This proves the inclusion \( \mathcal{C}_{\mathcal{C}_1, \mathcal{C}_2} \supseteq \mathcal{C}_{1,2} \). To prove the opposite inclusion, pick a saturated pair \( (A', G') \) belonging to \( \mathcal{C}_{\mathcal{C}_1, \mathcal{C}_2} \). Again, we distinguish two cases:

3. Case \( (A', G') \) is compatible. In this case we can take \( E = A' \), \( M_1 = G_1 \), and \( M_2 = G_2 \). By definition of \( \mathcal{C}_{\mathcal{C}_1, \mathcal{C}_2} \), we get \( M_1 \times M_2 \models^u (A', G'), M_2 \times E \models^u (A_1, G_1), \) and \( M_1 \times E \models^u (A_2, G_2) \),
which translates as $G_1 \cap G_2 \subseteq G'$, $G_2 \cap A' \subseteq A_1$, and $G_1 \cap A' \subseteq A_2$. Thus $(A', G')$ belongs to $C_{1,2}$.

4. Case $(A', G')$ is incompatible. In this case, the last two conditions on the right most column of (5.8) are trivially satisfied. The same argument as for the case 3 shows that the first condition on the right most column of (5.8) is satisfied. Thus, again, the considered pair $(A', G')$ belongs to $C_{1,2}$.

This proves the inclusion $C_{\mathcal{G}_1, \mathcal{G}_2} \subseteq C_{1,2}$, and thus completes the proof of (5.8).

In a second step, by invoking (4.1) and (4.2) it remains to calculate $\bigwedge C_{\mathcal{G}_1, \mathcal{G}_2}$, using (5.8). We first prove that formulas (5.7) yield a saturated contract, that is: we must prove $\neg A \subseteq G$. This follows from the following inclusions: $\neg A = \neg(A_1 \cap A_2) \cap (G_1 \cap G_2) = (\neg A_1 \cup \neg A_2) \cap (G_1 \cap G_2) \subseteq (G_1 \cup G_2) \cap (G_1 \cap G_2) = G_1 \cap G_2 = G$. Next, we note that $G$ is minimal among the $G'$ satisfying the first condition of (5.8). Finally, the maximal $A'$ satisfying $A' \cap G_2 \subseteq A_1$ is $A_1 \cup \neg G_2$. So the maximal $A'$ satisfying the second and third conditions of (5.8) is

\[
(A_1 \cup \neg G_2) \cap (A_2 \cup \neg G_1) = (A_1 \cap A_2) \cup (A_1 \cap \neg G_1) \cup (A_2 \cap \neg G_2) \cup (\neg G_2 \cap \neg G_1) = (A_1 \cap A_2) \cup \neg G_1 \cup \neg G_2 \cup (\neg G_2 \cap \neg G_1) = (A_1 \cap A_2) \cup (\neg G_1 \cap G_2)
\]

where the second equality follows from the fact that the pairs $(A_i, G_i)$, $i = 1, 2$ are saturated. This finishes the proof of the theorem. As a byproduct of this proof we get that Axiom 2 holds.

**Theorem 5.3.** Axiom 4 of the meta-theory holds.

**Proof.** It is an immediate consequence of Theorem 4.7 applied to saturated $A/G$ contracts, with $\subseteq$ being the inclusion of assertions seen as sets of behaviors, $G$ and the silent component $\{\varepsilon\}$ being the maximal and the minimal implementations of $(A, G)$, respectively.

Hence contract composition is associative. No Least Upper Bound and no quotient are known for $A/G$ contracts.
Comment 5.4 (regarding saturated contracts). As the reader has noticed, getting saturated contracts is important in applying this contract algebra. This seems to require being able to compute with unions and complements of assertions. In fact, we only need to be able to compute the operation \( G \cup \neg A \), which we like to interpret as the entailment \( A \Rightarrow G \). More precisely, it is enough to have a tool able to synthesize models for formulas of the form \( A \Rightarrow G \), where: \( A \) is a formula or a conjunction of formulas, \( G \) is a formula or a conjunction of formulas, or, recursively, \( G \) has the form \( A \Rightarrow G \). As we shall see in Chapter 6, it turns out that the Moore Interfaces, the simplest form of Synchronous Component Interfaces proposed in [82], provide a way of computing this entailment, for a restricted class of \( A/G \) contracts. □

We finish this section by observing that the two contracts \( C_1 \) and \( C'_1 \) of Section 2.2.1 satisfy \( C'_1 \leq C_1 \) according to the theory of this section: guarantees are identical but assumptions are relaxed.

5.2 Dealing with variable alphabets

Since contracts aim at capturing incomplete designs, we cannot restrict ourselves to a fixed alphabet of variables — it is not known in advance what the actual alphabet of variables of the complete design will be. Thus the simple variants of Section 5.1 has little practical relevance and we must address variable alphabets. Therefore, components will now be pairs \( M = (V_M, P_M) \), where \( V_M \) is the alphabet of variables of \( M \) and \( P_M \) is an assertion over \( V_M \). Similarly, contracts are tuples \( C = ((V_A, A), (V_G, G)) \), where assumptions \( A \) and guarantees \( G \) are assertions over alphabets of variables \( V_A \) and \( V_G \).

Key to dealing with variable alphabet of variables is the operation of alphabet equalization that we introduce next. For \( P \) an assertion over the alphabet of variables \( V \) and \( V' \subseteq V \), we consider its projection \( \text{pr}_{V'}(P) \) over \( V' \), which is simply the set of all restrictions, to \( V' \), of all behaviors belonging to \( P \). We will also need the inverse projection \( \text{pr}_{V''}^{-1}(P) \), for \( V'' \supseteq V \), which is the set of all behaviors over \( V'' \) projecting to \( V \) as behaviors of \( P \). For \((V_i, P_i), i = 1, 2\), we call alphabet equalization of \((V_1, P_1)\) and \((V_2, P_2)\) the two assertions \((V, \text{pr}_{V'}^{-1}(P_i)), i = 1, 2\) where \( V = V_1 \cup V_2 \).
This being defined, all operations and relations introduced in Section 5.1 are extended to the case of variable alphabet by 1) applying alphabet equalization to the involved assertions, and, 2) reusing the operation or relation as introduced in Section 5.1.

**Comment 5.5 (A practical pitfall of A/G contracts with variable alphabet).** As pointed out in [40], this generalization yields a contract theory that is a valid instance of the meta-theory (up to the missing quotient). It is not fully satisfactory from the practical standpoint, however. The reason is that the conjunction of two contracts having disjoint alphabets of variables always yields a trivial assumption \( t \), which is very demanding — any environment must be accommodated — and does not reflect the intuition. See Section 8.5 for a different contract framework, not suffering from this drawback. □

### 5.3 Abstractions

For concepts and undefined notations, the reader is referred to Section 4.7. We assume two sets \( M_c \) and \( M_a \) of concrete and abstract components. \( M_c \) and \( M_a \) are ordered by set inclusion. We assume a Galois connection \((\alpha, \gamma)\) between \( M_c \) and \( M_a \). Concrete and abstract A/G contracts are pairs \( C_c = (A_c, G_c) \) and \( C_a = (A_a, G_a) \) of concrete/abstract assumptions and guarantees. \( E_c \models C_c \) if \( E_c \) satisfies \( A_c \), i.e., \( E_c \subseteq A_c \). \( M_c \models C_c \) if \( M_c \cap A_c \) satisfies \( G_c \), i.e., \( M_c \cap A_c \subseteq G_c \). Since components are sets of behaviors, the intuitive choice for \( \subseteq_{c/a} \) is set inclusion. This complies with Axiom 5. Using (4.17) yields, for two concrete and abstract A/G contracts \( C_c = (A_c, G_c) \) and \( C_a = (A_a, G_a) \),

\[
\overline{\alpha}(C_c) = \overline{\alpha}(A_c, G_c) = (\alpha(A_c), \alpha(G_c)) \quad (5.10)
\]

Formula (5.10) requires putting contracts in saturated form, see Remark 5.4.
5.4. Observers

It remains to explain how to construct a Galois connection for components defined as sets of behaviors. To be able to apply Theorem 4.10, we are interested in knowing if $\gamma$ is (sub)-multiplicative. It is shown in [45] that predicate abstraction applied to both the initial condition and the transition relation, defines a Galois connection over components in which concretization $\gamma$ is multiplicative, so that Theorem 4.10 applies. Recall that predicate abstraction works as follows—we explain it for transition relations. Select an arbitrary finite set $(P_i)_{i \in I}$ of predicates over $2^{D_{c} \times D_{c}}$. For a concrete transition relation $R \subseteq 2^{D_{c} \times D_{c}}$, $P_i(R)$ returns true or false, depending on whether $R$ satisfies this predicate or not. The abstraction of $R$ is then defined as $\alpha(R) = (P_i(R))_{i \in I} \in \mathbf{Bool}^I = \text{def} \ M_a$. It is seen that $\alpha$ is a complete $\sqcap_c$-morphism, hence a unique concretization $\gamma$ can be canonically associated with it, making $(\alpha, \gamma)$ a Galois connection [93, 202]. Equipping the abstract domain $M_a$ with the product $\times_a = \text{def} \ \sqcap_a$, where the infimum $\sqcap_a$ refers to the product order on the abstract domain $M_a = \mathbf{Bool}^I$, ensures that the so defined $\gamma$ is multiplicative. See [45] for details.

Thus, abstractions exist for $A/G$ contracts. They allow for proving implementation and environment relations, for proving consistency and compatibility, and offer some modularity in checking these properties. It is of interest to complement abstractions with observers we introduce next. Alike testing, observers allow disproving the above properties. The two semi-decision procedures are thus complementary.

5.4 Observers

Observers are often used to complement Simulink [168] or Scade [206, 47] models with on-line property monitoring. The observers we propose here are a variation of this existing practice, to match the discipline of contract based design. Observers relate to the widespread technique of testing. A test consists in executing a program and assessing whether or not the outcome satisfies the expected property. The program generally possesses several executions (it may itself involve internal non-determinism or its outcome may depend on the inputs submitted to it). Therefore, the verdict of the test may be positive (the property holds for a given execution) or negative (a given execution violates the property). Getting a negative verdict shows that the
property may not be met, but getting a positive verdict proves nothing (an-
other unexplored execution may violate the property). This is referred to as a
semi-decision procedure.

We now proceed to formalizing observers in the context of synchronous
A/G contracts. Given a behavior $\sigma$ and two assertions $P$ and $Q$ (two sets
of behaviors) over the alphabet $V$ of variables,

\[
say that b(\sigma, Q, P) = \tau \text{ if } \sigma \in P \cup \neg Q, \text{ and otherwise say that } b(\sigma, Q, P) = \false.
\]

(5.11)

To capture the notion of success or failure of a test suite, we lift this definition
to a finite set $S$ of behaviors over $V$ by stating:

\[
say that b(S, Q, P) = \tau \text{ iff } b(\sigma, Q, P) = \tau \text{ holds for every } \sigma \in S, \text{ and otherwise say that } b(S, Q, P) = \false.
\]

(5.12)

Given a component $M$, we denote by $S_M$ a finite subset of its behaviors —
the intent is that $S_M$ collects the tests actually performed on the system. Given
two assertions $Q, P$, a component $M$ and a finite subset $S_M$ of its behaviors,
if $b(S_M, Q, P) = \false$, then $M$ violates the assertion $P$ when put in the context $Q$.
Nothing can be said, however, in case $b(S_M, Q, P) = \tau$, since it may be that
$b(S'_M, Q, P) = \false$ holds for another set $S'_M$ and not all finite subsets of $M$ can
be explored.

**Definition 5.5.** An *observer* of contract $C = (A, G)$ (possibly not in satu-
rated form) is a pair $(b_E^C, b_M^C)$ of maps called *verdicts*:

\[
b_E^C(E) = \defeq \sigma_E \rightarrow b(\sigma_E, \Omega, A)
b_M^C(M) = \defeq \sigma_M \rightarrow b(\sigma_M, A, G)
\]

(5.13)

where $\sigma_E$ and $\sigma_M$ range over the set of all behaviors of $E$ and $M$, respectively,
$\Omega$ denotes the set of all runs, and the boolean map $b$ is defined in (5.11). $\square$

Since in general only a finite subset of all behaviors can be run, only
semi-decisions can be obtained and observers can only be used to *disprove*
the legality of an environment or an implementation.

Observers do not require the contract to be in saturated form. In fact, by
construction,

\[
\text{for } \overline{C} = (A, G \cup \neg A), \text{ we have } b_E^\overline{C} = b_E^C \text{ and } b_M^\overline{C} = b_M^C
\]

(5.14)
5.4. Observers

Consequently, for deriving all the subsequent formulas, with no loss of generality, we can always assume that the involved contracts are in saturated form.

Some relations between contracts require decision procedures, semi-decisions being insufficient. In particular, observers cannot be used to prove nor disprove consistency or compatibility. Neither can they prove nor disprove contract refinement.

Nevertheless, the following compositional formulas yield a pair of observers for the conjunction, where \( \text{pr}_{V'}(\sigma') \) denotes the projection, to subalphabet \( V \subseteq V' \), of behavior \( \sigma' \) defined over alphabet \( V' \), see Section 5.2, and \( V = V_1 \cup V_2 \):

\[
\begin{align*}
  b_{e,1 \land e_2}^e(E) &= \text{def } \sigma_E \rightarrow b(\text{pr}_{V_1}(\sigma_E), \Omega, A_1) \lor b(\text{pr}_{V_2}(\sigma_E), \Omega, A_2) \\
  b_{e,1 \land e_2}^m(M) &= \text{def } \sigma_M \rightarrow b(\text{pr}_{V_1}(\sigma_M), A_1, G_1) \land b(\text{pr}_{V_2}(\sigma_M), A_2, G_2)
\end{align*}
\]

(5.15)

which, by using the boolean algebra of boolean functions, rewrites:

\[
\begin{align*}
  b_{e,1 \land e_2}^e &= \text{def } \overline{b_{e,1}^e} \lor \overline{b_{e,2}^e} \\
  b_{e,1 \land e_2}^m &= \text{def } \overline{b_{e,1}^m} \land \overline{b_{e,2}^m}
\end{align*}
\]

(5.16)

where, for \( i = 1, 2 \),

\[
\begin{align*}
  \overline{b_{e_i}^e}(E) &= \text{def } \sigma_E \rightarrow b(\text{pr}_{V_i}(\sigma_E), \Omega), \text{ and} \\
  \overline{b_{e_i}^m}(M) &= \text{def } \sigma_M \rightarrow b(\text{pr}_{V_i}(\sigma_M), A_i, G_i)
\end{align*}
\]

(5.17)

The construction of observers for contract composition is similar. Using again (5.17), the following formulas yield observers for the parallel composition of contracts:

\[
\begin{align*}
  b_{e_1 \oplus e_2}^e &= (\overline{b_{e_1}^e} \land \overline{b_{e_2}^e}) \lor \neg (\overline{b_{e_1}^m} \land \overline{b_{e_2}^m}) \\
  b_{e_1 \oplus e_2}^m &= \overline{b_{e_1}^e} \land \overline{b_{e_2}^e}
\end{align*}
\]

(5.18)

Observers for \( e_1 \land e_2 \) or \( e_1 \oplus e_2 \) must be applied according to the following procedure — we only explain it for implementations, but the same holds for environments:

**Procedure 1** (modular observers).

1. Select a finite set \( S_M \) of behaviors for the implementation \( M \) under monitoring;
2. Project every behavior $\sigma_M \in S_M$ over each subalphabet $V_1$ and $V_2$, which yields two local behaviors $\sigma_1$ and $\sigma_2$;

3. Submit $\sigma_1$ and $\sigma_2$ to local observers, respectively;

4. Fuse the results by using formulas (5.16) or (5.18) and conclude by using (5.12).

Note that $\sigma_1$ and $\sigma_2$ are “synchronized” in that merging them yields $\sigma_M$ back. Thus, the behaviors are produced globally, and then projected for submission to local observers. In this sense, only partial modularity is achieved. Is it possible to do better? More precisely,

is it possible to let the observers be applied independently (meaning that the tested behaviors are produced independently) and then conclude by simply fusing the outcomes of each observer?

We now discuss question (5.19) in more detail.

**Modular observers for the conjunction:** Suppose that $E$ and $M$ are a candidate environment and a candidate implementation for the contract $C = C_1 \land C_2$, and the two contracts $C_1$ and $C_2$ are developed by different teams.

Consider first the question (5.19) regarding the candidate implementation $M$. In the conjunction, the same implementation $M$ is used to generate behaviors for testing $C_1$, $C_2$, and $C_1 \land C_2$. If we find a behavior $\sigma$ of $M$ projected to the alphabet $V_1$ of $C_1$ (let us call this projection $\sigma_1$) that violates $C_1$, all the inverse projections of $\sigma_1$ (including $\sigma$, but not only) will violate $C_1 \land C_2$, hence we have modularity in testing implementations. To formalize this, for $i = 1, 2$, let $S_{E,i}$ be the finite set of behaviors to be submitted to $b_{C_i}^E (E)$ and let $S_{M,i}$ be the finite set of behaviors to be submitted to $b_{C_i}^M (M)$. Let $V_i$ be the set of variables of contract $C_i$. The set $V$ of variables of $M$ contains $V_1 \cup V_2$. Set

$$S_M = \text{def } \text{pr}^{-1}_V (S_{M,1}) \cup \text{pr}^{-1}_V (S_{M,2})$$

When testing implementations in the conjunction of contracts, we have by (5.15):

$$\exists \sigma \in S_M : b(\sigma, A, G) = F$$

$$\iff \exists i \in \{1, 2\}, \exists \sigma_i \in S_{M,i} : b(\sigma_i, A_i, G_i) = F$$

(5.20)
expressing that a failure of \( M \) to implement \( C \) is detected by the test set \( S_M \) if and only if \( M \) was detected violating \( C_1 \) or \( C_2 \) by the local tests performed independently: the answer to question (5.19) is yes for testing implementations in contract conjunction.

The same argument does not work for testing the environment \( E \), however, because the required counterpart of (5.20) should be the wrong statement

\[
\exists \sigma \in S_E : b(\sigma, \Omega, A) = \mathsf{f} \\
\iff \forall i \in \{1, 2\}, \exists \sigma_i \in S_{E,i} : b(\sigma_i, \Omega, A_i) = \mathsf{f}
\]

In (5.21), the right hand side is strictly weaker than the left hand side because the two behaviors \( \sigma_i \) selected by the existential quantifier may not lift to the same \( \sigma \) back. Still, the complement of (5.21) holds, namely:

\[
\exists \sigma \in S_E : b(\sigma, \Omega, A) = \mathsf{t} \\
\iff \exists i \in \{1, 2\}, \exists \sigma_i \in S_{E,i} : b(\sigma_i, \Omega, A_i) = \mathsf{t}
\]

which expresses that testing \( E \) as a candidate environment for \( C \) is inconclusive based on the test set \( S_E \) if and only if testing \( E \) as a candidate environment for \( C_1 \) or \( C_2 \) was inconclusive based on the local tests performed independently. This kind of modularity is less relevant since the real objective of testing is to discover bugs.

**Modular observers for the parallel composition:** What about question (5.19) for contract composition? In composition, we have now three components \( M_1, M_2, \) and \( M_1 \times M_2 \) that enter into the play, for testing \( C_1, C_2, \) and \( C_1 \otimes C_2, \) respectively. If we find a local behavior \( \sigma_1 \) of \( M_1 \) that violates \( C_1, \) it still does not mean that we can find a behavior of \( M_1 \times M_2 \) that violates \( C_1 \otimes C_2, \) because the local behavior \( \sigma_1 \) may never be able to occur, due to the restrictions imposed by the composition \( M_1 \times M_2. \) Formally, (5.20) is wrong for contract composition, for testing both implementations and environments.

### 5.5 Asynchronous dataflow A/G contracts

Sections 5.1–5.4 together define a framework of synchronous A/G contracts in which components are synchronous systems. In this section, we sketch
a variant of asynchronous Assume/Guarantee contracts where the components are (asynchronous) dataflow Kahn Process Networks. We obtain this by changing the concrete definition of what an assertion is:

\[ P \subseteq (V \mapsto D^* \cup D^\omega) \]  

(5.23)

Compare (5.23) with (5.1). In both cases, assertions are sets of behaviors. In (5.1) behaviors were finite or infinite sequences of reactions, which consist in the assignment of a value to each symbol of the alphabet of variables. In contrast, in (5.23), behaviors are tuples of finite or infinite flows, one for each symbol of the alphabet of variables. Definition (5.1) for assertions correspond to the synchronous model of computation, whereas (5.23) corresponds to the Kahn Process Network type of model [167, 204]. The material of Sections 5.1–5.4 can be adapted to this new model of component composition, thus yielding a framework of asynchronous dataflow Assume/Guarantee contracts.

5.6 A/G contracts for Cyber-Physical systems

Cyber-Physical systems (CPS) possess several aspects (or viewpoints): the performed function, timing behavior and scheduling, energy consumption, safety/reliability, and more. Our objective in this section is to equip formalisms dedicated to CPS [74] such as Simulink [168] or Modelica [133] with A/G contracts capturing the function being specified or implemented. Both formalisms capture hybrid systems, or, equivalently, systems possessing multiple modes where the dynamics in each mode involves Ordinary Differential Equations (ODE) or Differential Algebraic Equations (DAE), for Modelica.\(^2\) The only relevant information for us is that a component is characterized by its set of behaviors, i.e., the set of all its possible executions.\(^3\)

Formally, a behavior for a hybrid system is a partial function

\[ \mathbb{R}_+ \rightarrow V \rightarrow D \]  

(5.24)

\(^2\)ODE have the form \(x' = f(x, u)\) where \(u\) is the input continuous time signal, \(x\) is the state, and \(x'\) denotes the time derivative of \(x\). DAE are constraints of the form \(f(x, x', u) = 0\) that relate the two signals \(u\) and \(x\). DAE are not oriented, i.e., they are not input/output functions.

\(^3\)Control and numerical engineers use the term “simulation” instead of “execution” but we prefer to stick to the latter because the former has a different meaning in the context of this work.
where $\mathbb{R}_+ = [0, +\infty)$ is the set of non-negative real numbers figuring time,
$V$ is the considered set of variables, and $D$ is the desired domain (which should typically include reals, integers, and booleans). A behavior is therefore a function that, for each instant in time, associates a value to each of the system variables. Actually, such behaviors can be used as a domain to express the semantics of languages such as Simulink or Modelica. To summarize,

assertions for CPS are simply sets of behaviors according to (5.24); the trivial behavior $\varepsilon$ is the behavior having empty domain and the components are all the assertions containing $\varepsilon$. (5.25)

Being entirely formulated in terms of abstract sets of behaviors, the A/G contract framework developed in Sections 5.1–5.4 extends with no change to the situation defined by (5.24,5.25). Axioms 1–4 of the meta-theory hold. Observers can be used and provide the due discipline of testing, for such contracts.

The above abstract class (5.25) of components is larger than the subclass of components that can be specified through Simulink or Modelica models: a Simulink model specifies the components whose set of behaviors is the set of all executions of the Simulink model when the inputs range over the set of possible input trajectories; a Modelica model specifies the components whose behaviors are solutions of the Modelica model. Simulink and Modelica models impose smoothness constraints, so not every component in the sense of (5.25) can be specified with these formalisms. Still, the set of behaviors (5.25) can be used as a domain to express the semantics of these more restricted models. Therefore, embedding the class of Simulink or Modelica models in the larger class (5.25) allows us to invoke all the properties shown for the meta-theory and apply them to actual Simulink or Modelica models.

5.7 Discussion

A/G contracts are a family of instantiations of our meta-theory. This family is flexible in that it can accommodate different models of communication — synchronous, asynchronous, continuous real-time. A/G contracts are an adequate framework for use in requirements capture. Indeed, requirements are naturally seen as assertions and, when categorizing requirements into assumptions (specifying the context of use of the system under development)
and guarantees (what the system offers), formalizing the resulting set of requirements as an A/G contract seems very natural.

The framework of A/G contracts does not handle variable alphabets in a satisfactory way for the refinement and the contract conjunction, see Comment 5.5 in Section 5.2.

One difficulty of this framework is the need for performing contract saturation. The Moore Interfaces and their variations (Section 6.3) provide an effective game-based approach that implements contract saturation. Remember that this framework imposes restrictions on the class of A/G contracts considered.

For general modeling frameworks involving complex data types (e.g., reals), using observers or abstract interpretation can mitigate the difficulties in making the operations of the contract algebra effective.

5.8 Bibliographical note

By explicitly relying on the notions of Assumptions and Guarantees, A/G contracts are intuitive, which makes them appealing for the engineer. In A/G contracts, Assumptions and Guarantees are just properties. The typical case is when these properties are languages or sets of traces, which includes the class of safety properties [173, 84, 196, 23, 89]. A/G contracts were advocated by the SPEEDS project [40, 46]. They were further experimented in the framework of the CESAR project [99]. The theory developed in [40] turns out to be closest to this presentation. The presentation developed in this monograph clarifies the design choices in A/G contract theories.

The Moore Interfaces proposed in [82] can be seen as an elegant and operational specialization of A/G contracts, in which the operation of contract saturation is replaced by the consideration of a game associated to the interface. In fact, the Moore Interfaces are not an interface theory in the sense of section 8 but are rather related to this class of A/G contracts. For this specialization to be possible, three restrictions are needed: (i) the assumption A and the guarantee G are both specified as transition systems; (ii) all variables must have finite domain; and, (iii) more importantly, the assumption A and the guarantee G act on disjoint sets of variables, controlled by the environment and the component respectively.
Interface Input/Output automata were proposed in [176] as a pair of two i/o-automata acting as assumption and guarantee. A comparison with interface automata is given.

Inspired by [176], another form for A/G contract was proposed by [137, 138, 141] when designs are expressed using the BIP programming language [56, 243]. To achieve separate development of components, and to overcome the problems that certain models have with the effective computation of the operators, the authors avoid using parallel composition $\otimes$ of contracts. Instead, they replace it with the concept of circular reasoning, which states as follows in its simplest form: if design $M$ satisfies property $G$ under assumption $A$ and if design $N$ satisfies assumption $A$, then $M \times N$ satisfies $G$. When circular reasoning is sound, it is possible to check relations between composite contracts based on their components only, without taking expensive compositions. In order for circular reasoning to hold, the authors devise restricted notions of refinement under context and show how to implement the relations in the contract theory for the BIP framework. Compatibility is not addressed and this proposal does not consider conjunction.

A/G contracts are proposed in [87] for finite traces of interface automata. Safety and progress for possibly nondeterministic automata are addressed by characterizing a component through observable, inconsistent (raising an exception), and quiescent (reaction termination) traces. The satisfaction relations for environments and implementations are adjusted to account for this more precise characterization of components. Refinement, conjunction, disjunction, parallel composition, and quotient are proposed: this development is therefore remarkably comprehensive. An interesting comparison with [176] is developed using an illustration example.

The automatic generation of observers for A/G contracts has been proposed in the work [129, 128], where assertions are specified using a declarative pattern-based language. A set of monitors is then generated and implemented in the Simulink framework to observe the underlying system execution and flag behaviors that violate either the assumptions or the guarantees. The method seems suitable for analyzing the implementation relation, while consistency and compatibility are only analyzed for closed systems.
Regarding extensions, a notion of contract for real-time interfaces is proposed in [54]. Sets of tasks are associated to components which are individually schedulable on a processor. An interface for a component is an $\omega$-language containing all legal schedules. Schedulability of a set of components on a single processor then corresponds to checking the emptiness of their intersection. The interface language considered is expressive enough to specify a variety of requirements like periodicity, the absence or the presence of a jitter, etc. An assume/guarantee contract theory for interfaces is then developed where both assumptions and guarantees talk about bounds on the frequency of task arrivals and time to completions. Dependencies between tasks can also be captured. Refinement and parallel product of contracts are then defined exactly as in the Speeds generic approach. In the same direction, A/G contracts were proposed in [246, 231, 232, 245] for real-time scheduling problems, where tasks and their data dependencies, and resources, must be handled. See also Chapter 9.

In [207], a platform-based design methodology that uses A/G analog contracts is proposed to develop reliable abstractions and design-independent interfaces for analog and mixed-signal integrated circuit design. Horizontal and vertical contracts are formulated to produce implementations by composition and refinement that are correct by construction. The effectiveness of the methodology is demonstrated on the design of an ultra-wide band receiver used in an intelligent tire system, an on-vehicle wireless sensor network for active safety applications. A similar approach using A/G contracts was employed more recently for the assessment of dependability in service-oriented architectures [96].

A/G contracts have been extended to a stochastic setting by Delahaye et al. [108, 109, 110]. In this work, the implementation relation becomes quantitative. More precisely, implementation is measured in two ways: reliability and availability. Availability is a measure of the time during which a system satisfies a given property, for all possible runs of the system. In contrast, reliability is a measure of the set of runs of a system that satisfy a given property. Following the lines of the contract theories presented earlier, satisfaction is assumption-dependent in the sense that runs that do not satisfy the assumptions are considered to be “correct”; the theory supports refinement, structural
composition and logical conjunction of contracts; and compositional reasoning methods have been proposed, where the stochastic or non-stochastic satisfaction levels can be budgeted across the architecture: For instance, assume that implementation $M_i$ satisfies contract $C_i$ with probability $\alpha_i$, for $i = 1, 2$, then the composition of the two implementations $M_1 \times M_2$ satisfies the composition of the two contracts $C_1 \otimes C_2$ with probability at least $\alpha_1 + \alpha_2 - 1$.

**Observers:** Observers, being related to the wide area of software and system testing, have been widely studied. A number of existing technologies support the design of observers and we review some of them now. First of all, observers are related to the widely explored area of so-called IOCO-testing. The work [95] bridges the gap between this area and observers for contracts by re-considering compositional testing in view of contract composition.

**Synchronous languages** [38, 142, 44] are a formalism of choice in dealing with observers. The family of Synchronous Languages comprises mainly the imperative language Esterel [132, 119] and the dataflow languages Lustre [206] and Signal [216]. The family has grown with several children offering statecharts-like interfaces and blending dataflow and statechart-based styles of programming, such as in Scade V6. Synchronous languages support only systems governed by discrete time, not systems with continuous time dynamics (ODEs). They benefit from a solid mathematical semantics. As a consequence, executing a given program always yields the same results (results do not depend on the type of simulator). The simulated or analysed program is identical to the code for embedding. Thanks to these unique features, specifications can easily be enhanced with timing and/or safety viewpoints. The widely used Simulink/Stateflow tool by The Mathworks offers similar features. One slight drawback is that its mathematical semantics is less firmly defined (indeed, results of executions may differ depending on the code executed: simulation or generated C code). On the other hand, Simulink supports continuous time dynamics in the form of systems of interconnected ODEs (Ordinary Differential Equations), thus supporting the modeling of the physical part of the system. Using Simulink, possibly enhanced with SimScape.

\url{http://www.esterel-technologies.com/products/scade-suite/}
\url{https://www.mathworks.com/products/simulink.html}
\url{https://www.mathworks.com/products/simscape.html}
allows for including physical system models in observers, e.g., as part of the system environment. The same comment holds regarding Modelica.\footnote{https://www.modelica.org/} Actually, observers have been proposed and advocated in the context of Lustre and Scade [143, 144, 145], Esterel [60], and Signal [197, 198]. More precisely, Scade advocates expressing tests using Scade itself. Tests can then easily be evaluated at run time while executing a Scade program. To conclude, observe that synchronous languages and formalisms discussed in this section are commercially available and widely used.

Another good candidate for expressing observers is the Property Specification Language (PSL). PSL is an industrial standard [210, 122, 121] for expressing functional (or behavioral) properties targeted mainly to digital hardware design. We believe that PSL is indeed very close to several, less established but more versatile formalisms based on restricted English language that are used in industrial sectors other than digital hardware, e.g., in aeronautics, automobile, or automation. Consider the following property:

“\textit{For every sequence that starts with an a immediately followed by three occurrences of b and ends with a single occurrence of c, d holds continuously from the next step after the end of the sequence until the subsequent occurrence of e.}”

This property is translated into its PSL version

\[
\{ [*];a;b[*3];c \} \Rightarrow (d \text{ until! e})
\]

PSL is a well-suited specification language for expressing functional requirements involving sequential causality of actions and events. Although we are not aware of the usage of PSL in the particular context of contract-based design, we mention the tool FoCS [6] that translates PSL into checkers that are attached to designs. The resulting checker takes the form of an observer, if the PSL specification is properly partitioned into assumption and guarantee properties. More recently, PSL was also used for the generation of transactors that may adapt high-level requirements expressed as transaction-level modules to the corresponding register-transfer implementation [25, 26]. It follows that the existing tool support for PSL makes this specification language suitable
in the contract-based design using observers. We note that the availability of formal analysis tools allows the design to be checked exhaustively—this is, of course, at the price of restrictions on data types. Another benefit in using PSL as an observer-based interface formalism is an existing methodology for user-guided automated property exploration built around this language [220, 57], that is supported by the tool RATSY [58]. As previously stated, PSL is built on top of LTL and regular expressions. One can thus express liveness properties in PSL, which are not suitable for online monitoring. There are two orthogonal ways to avoid this potential issue: 1) restricting the PSL syntax to its safety fragment; or 2) adapting the PSL semantics to be interpreted over finite traces [123]. A survey of using PSL in runtime verification can be found in [121].

**Features of our presentation:** This presentation of A/G contracts is new in many respects. For the first time, it is cast into the meta-theory of contracts, with the advantage of clarifying the definition of refinement and parallel composition of contracts — this involved some hand waving in the original work [40]. This presentation of A/G contracts is complemented by a specialization for real-time scheduling in Chapter 9 with a corresponding application case in the context of AUTOSAR developed in Chapter 11.
6

Synchronous Moore Interfaces and A/G Contracts

6.1 Introduction

In a landmark paper [82], Synchronous Interfaces with the special case of Moore Interfaces, were introduced. Two verbatims from [82] (modulo notations) are reproduced below. These requirements for an interface theory suggest that synchronous interfaces should obey the meta-theory. While reading the above reference in an attempt to properly discussing it, we observed that the game associated to the composition of Moore interfaces seemed to solve the saturation operation on A/G contracts: \((A, G) \mapsto (A, G \cup \neg A)\), see (5.6) in Section 5.1. We thought that this observation was worth further investigations, which lead to this chapter in which we show that this guess was indeed correct. The contributions of this chapter are the following:

1. We show that the Moore Game of [82] yields an effective algorithm for performing the saturation operation \((A, G) \mapsto (A, G \cup \neg A)\).

2. We clarify the correspondence between A/G contracts and Moore Interfaces. It turns out to be almost perfect. The only missing feature of the alternating refinement of Moore Interfaces is the proper consideration of legal environments, which has consequences for the parallel composition of Moore Interfaces as well.
3. We propose a slight adjustment of the Moore Interfaces that match A/G contracts (and thus the meta-theory).

Two verbatims from [82] are reproduced below:

In the study of compatibility, game-based approaches quantify inputs existentially, and outputs universally. When two interfaces $C_1$ and $C_2$ are composed, their composition may have illegal states, where one component emits outputs that are illegal inputs for the other one. Yet, $C_1$ and $C_2$ are considered compatible as long as there is some input behavior that ensures that, for all output behaviors, the illegal states are avoided: in other words, $C_1$ and $C_2$ are compatible if there is some environment in which they can be used correctly together. In turn, the input behaviors that ensure compatibility constitute the legal behaviors for the composition $C_1 \otimes C_2$: when composing component models, both the possible output behaviors, and the legal input behaviors, are composed.

The game view leads to an alternating view of refinement: a more detailed interface $C_2$ refines an abstract interface $C_1$ if all legal inputs for $C_1$ are also legal for $C_2$, and if, when $C_1$ and $C_2$ are subject to the same legal inputs, $C_2$ generates output behaviors that are a subset of those of $C_1$. This definition ensures that, whenever $C_2 \preceq C_1$, we can substitute $C_2$ for $C_1$ in every design without creating any incompatibility: in the game view, substitutivity of refinement holds.

6.2 An illustration example for Moore Interfaces

To give the intuition behind Moore Interfaces, we reproduce the following example, borrowed verbatim from the thesis of Arindam Chakrabarti [80]. It is shown in Figure 6.1.

The guarded-command syntax used in this figure is derived from the one of reactive modules [12] and Mocha [15]; input atoms describe the input assumptions, and the output atoms describe the output behavior. When more
than one guard is true, the command is selected nondeterministically. Input
variables not mentioned by the command are updated nondeterministically.

```plaintext
interface Counter
output q0, q1: bool;
input cl: bool;
input atom
init
[] true -> cl := nondet
update
[] true -> cl := nondet
endatom
output atom
init
[] true -> q0 := 1; q1 := 1
update
[] cl -> q1if1 := 1; q0if1 := 0
[] ~cl & q1 & ~q0 -> q1if1 := 1; q0if1 := 0
[] ~cl & ~q1 & q0 -> q1if1 := 0; q0if1 := 1
[] ~cl & ~q1 & ~q0 -> q1if1 := 1; q0if1 := 1
endatom
end interface
```

```plaintext
interface Adder
input q0, q1: bool; di: [0..7];
output do: [0..7];
input atom
init
[] true -> q0 := 1
[] true -> q1 := 1
update
[] true -> q0 := 1
[] true -> q1 := 1
[] q0 & q1 -> do := di + 1
[] ~q0 & q1 -> do := di - 1
endatom
end interface
```

**Figure 6.1:** A counter (left) and an adder (right) modeled as Moore interfaces.

We illustrate the features of Moore interfaces by modeling a simple example: a ±1 adder driven by a binary counter. The adder `Adder` has two control inputs `q0` and `q1`, data inputs `i7,...,i0`, and data outputs `o7,...,o0`. When `q0 = q1 = 1`, the adder leaves the input unchanged: the next value of `o7,...,o0` is equal to `i7,...,i0`. When `q0 = 0` and `q1 = 1`, the next outputs are given by `[o'7,...,o'0] = [i7,...,i0] + 1 mod 2^8`, where primed variables denote the values at the next clock cycle, and `[o'7,...,o'0]` is the integer encoded in binary by `o'7,...,o'0`. Similarly, when `q1 = 0` and `q0 = 1`, we have `[o'7,...,o'0] = [i7,...,i0] - 1 mod 2^8.

The adder is designed with the assumption that `q0` and `q1` are not both 0; hence, the input transition relation of `Adder` states that `q0'q1' ≠ 0`. In order to cycle between adding 0, +1, −1, the control inputs `q0` and `q1` are connected to the outputs `q1` and `q0` of a two-bit count-to-zero counter `Counter`. The counter has only one input, `cl`: when `cl = 0`, then `q0'q1' = 11; otherwise, `[q1'q0'] = [q1,q0] - 1 mod 4.

When the counter is connected to the adder, the joint system can take a transition to a state where `q1,q0 = 00`, violating the adder’s input assumptions. In spite of this, the counter and the adder are compatible, since there is a way to use them together: to avoid the incompatible transition, it suffices to assert `cl = 0` early enough in the count-to-zero cycle of the counter. To
reflect this, when we compose Counter and Adder, we synthesize for their composition Counter $\times$ Adder a new input assumption, that ensures that the input assumptions of both Counter and Adder are satisfied.

To determine the new input assumption, we solve a game between Input, which chooses the next values of $cl$ and $i_7, \ldots, i_0$, and Output, which chooses the next values of $q_0, q_1$, and $o_7, \ldots, o_0$. The goal of Input is to avoid a transition to $q_1q_0 = 00$. At the states where $q_1q_0 = 01$, Input can win if $cl = 0$, since at the next clock cycle we will have $q_0'q_1' = 11$; but Input cannot win if $cl = 1$. By choosing $cl' = 0$, Input can also win from the states where $q_1q_0 = 10$. Finally, Input can always win from the states where $q_1q_0 = 11$, for all $cl'$. Thus, we associate with Counter $\times$ Adder a new input assumption encoded by the transition relation requiring that whenever $q_1q_0 = 10$, then $cl' = 0$. The input requirement $q_1q_0 = 00$ of the adder gives rise, in the composite system, to the requirement that the reset-to-1 occurs early in the count-to-zero cycle of the counter.

So far this was verbatim quote from [80]. This text illustrates the intuition for how composition works for Moore Interfaces. Can we relate this to the composition of A/G contracts?

Item 3 of Definition 5.4 states that, in the composition of A/G contracts, the overall assumption $A$ is discharged from what is already mutually guaranteed by the two contracts — this corresponds to the term $\cup \neg (G_1 \cap G_2)$. To parallel this with the discussion of the game associated with Moore Interfaces, the Input only checks what, in the raw product of the two machines, may lead to violating input assumptions of one interface. This expresses that the job of the game is to complement what is already natively offered by each interface.

Considering again the composition of A/G contracts, the remaining duty of the overall assumption $A$ is to ensure that input assumptions of both interfaces remain satisfied in the composition — referring to Item 3 of Definition 5.4, this corresponds to the term $A_1 \cap A_2$. But this is exactly what the game associated with Moore Interfaces finds, namely: “whenever $q_1q_0 = 10$, then $cl' = 0$” is the missing global property that inputs must satisfy in the composition of the two Moore interfaces.
This parallel suggests that there should be a tight relation between Moore Interfaces and A/G contracts. Formalizing this relation is the subject of this chapter.

6.3 A/G contract saturation via Moore Interfaces

In this section we develop the results announced in Comment 5.4 of Chapter 5 regarding contract saturation. We specialize our previous trace- or behavior-based framework of A/G contracts to a sub-case where the saturation operation can be made effective by using the Moore Interfaces.

6.3.1 Moore Interfaces and associated A/G contracts

We now assume that assertions $A$ and $G$ are defined via transition relations having a specific structure. We assume a disjoint copy $V'$ of the set $V$ of variables and call it the set of next variables. For $x \in V$, its counterpart in $V'$ is $x'$. For $P$ a predicate on $V$, we denote by $P'$ the predicate obtained by replacing in $P$ every $x \in V$ by $x' \in V'$. We next assume that each variable from $V$ has finite domain $D \cup \{\perp\}$, where $\perp$ means “absence”, see Section 5.1. Furthermore, a decomposition of $V$ is given into input and output variables:

$$V = V_{\text{in}} \uplus V_{\text{out}}.$$

We finally assume

- a predicate $I_A$ on $V_{\text{in}}$ and a predicate $T_A$ on $V \cup (V_{\text{in}})'$;
- a predicate $I_G$ on $V_{\text{out}}$ and a predicate $T_G$ on $V \cup (V_{\text{out}})'$.

Thus, predicates $I_A$ and $T_A$ control input variables, whereas predicates $I_G$ and $T_G$ control output variables.\footnote{In addition, [82] assumes some kind of satisfiability condition for these four predicates. We do not consider this assumption in our development.}

Following [82], we call Moore Interface the tuple

$$\mathcal{C} = (V, I_A, I_G, T_A, T_G).$$

Each Moore Interface defines an A/G contract $(A, G)$ where the two synchronous assertions $A$ (assumption) and $G$ (guarantee) are given by

$$A = \{ \sigma \mid \sigma(0) \models I_A \text{ and } \forall k. (\sigma(k), \sigma(k+1)) \models T_A \}$$

$$G = \{ \sigma \mid \sigma(0) \models I_G \text{ and } \forall k. (\sigma(k), \sigma(k+1)) \models T_G \}$$

(6.2)
where, as usual, symbol $|$ means “satisfies”. We now need to define what the components are, for this contract framework.

### 6.3.2 Components for Moore Interfaces

Since assumptions $A$ and guarantees $G$ are both specified as transition systems, it is natural to require that the underlying class $M$ of components consists of all transition systems on $V$ of the form

$$M = (V_{in}^M, V_{out}^M, I_M, T_M),$$

where $V = V_{in}^M \cup V_{out}^M$ is a decomposition of $V$ into input and output variables, the initial condition $I_M$ is a predicate over $V_{out}$, and the transition relation $T_M$ is a predicate over $V \cup (V_{out}')$. We assume the following conditions on predicates $I_M$ and $T_M$, where $[V/\bot]$ denotes the assignment of the value $\bot$ to every variable belonging to $V$ and similarly for $[V_{out}/\bot]$:

1. $[V/\bot]$ satisfies $I_M$; and
2. $\forall V . T_M[V_{out}/\bot]$ holds,

which means that $M$ is stuttering invariant. Note that, for an arbitrary pair $(I_M, T_M)$, the transformation

$$(I_M, T_M) \mapsto (I_M \lor [\forall v \in V : v = \bot], T_M \lor [\forall v' \in V_{out} : v' = \bot])$$

returns a pair satisfying (6.3). It is, however, a weakening of the original pair.

Two components $M_1$ and $M_2$ are called composable if $V_{out}^M \cap V_{out}^{M_2} = \emptyset$. The composition $M = M_1 \times M_2$ is given by

$$V_{out}^M = V_{out}^{M_1} \cup V_{out}^{M_2}, \quad V_{in}^M = V \setminus V_{out}^M, \quad I_M = I_{M_1} \land I_{M_2}, \quad T_M = T_{M_1} \land T_{M_2}$$

Observe that the so defined pair $(I_M, T_M)$ satisfies (6.3). The composition $\times$ defined by (6.5) is associative and commutative.

### 6.3.3 Computing the maximal environment and the maximal implementation

The authors of [82] associate, to a pair of Moore Interfaces, a certain two-player game and use it to define the parallel composition and compatibility
condition. In our development, we reuse a variation of this game to compute the most liberal environment and the most liberal implementation.

More precisely, to a Moore Interface $I$, we associate the two-player “Moore game” $\Gamma$ introduced next. Playing $\Gamma$ results in the construction of a certain behavior through its successive reactions. Each round of the game extends the current behavior by one more reaction. We borrow the description of the game $\Gamma$ from [82], while exchanging the roles of players $in$ and $out$.

**Definition 6.1 (Moore game $\Gamma$).** At each round of the game:

- Player $in$ chooses new values for the input variables $V^{in}$ according to $I_A$ at the first round, and then according to $T_A$;
- Simultaneously and independently, player $out$ chooses unconstrained new values for the output variables $V^{out}$.

Player $out$ wins if the resulting behavior belongs to $G$ defined in (6.2). $\Box$

The Moore game $\Gamma$ is an adaptation of the game introduced in [82] — the original game will be reintroduced in our context in Section 6.4.1, when discussing the compatibility between Moore Interfaces and their parallel composition. We closely adapt from [82] an iterative algorithm for computing, if it exists, the most liberal winning strategy for player $out$. This algorithm approximates iteratively:

- the predicate $C$ characterizing the set of states from which the player $out$ can win the game, and
- the most liberal winning transition relation.

Set $C_0 = \tau$ and, for $k \geq 0$:

\[
T_{k+1} = \forall(V^{in})'. T_A \Rightarrow (T_G \land C_k')
\]
\[
C_{k+1} = C_k \land \exists(V^{out})'. T_{k+1}
\]

(6.6)

Note that $T_{k+1}$ is a predicate on $V \cup (V^{out})'$ and $C_{k+1}$ is a predicate on $V$. The sequences of predicates $C_k$ and $T_k$ are non-increasing. Since all variables possess a finite domain, the convergence of $C_k$ and $T_k$ to their limits $C_\infty$ and
6.3. A/G contract saturation via Moore Interfaces

$T_\infty$ arises in finitely many steps and we have

$$
\begin{align*}
C_\infty &= \exists (V^{\text{out}})' \forall (V^{\text{in}})', [T_A \Rightarrow (T_G \land C_\infty')]
\end{align*}
$$

which expresses that $C_\infty$ represents the set of states from which player $\text{out}$ can win the game when setting the initial condition of $G$ to true. Hence,

- $I_* = \text{def} \ [I_A \Rightarrow I_G] \land C_\infty$ is the weakest initial condition that player $\text{out}$ must select;
- $T_* = \text{def} \ [C_\infty \Rightarrow T_\infty]$ is the most liberal transition relation for $\text{out}$ to win the game.

The following result is immediate:

**Lemma 6.1.** If $T_G$ satisfies (6.3-\(b\)), then the pair ($C_\infty, T_\infty$) satisfies (6.3). If, in addition, $I_G$ satisfies (6.3-\(a\)), then the pair ($I_*, T_*$) also satisfies (6.3). □

Reference [82] contains detailed implementation considerations regarding algorithm (6.6). If $(I_A, T_A)$ satisfies (6.3), then $\mathcal{C}$ is compatible and we can consider the component

$$
E_{\mathcal{C}} = \text{def} \ (V^{\text{out}}, V^{\text{in}}, I_A, T_A).
$$

If player $\text{out}$ can win, i.e., $I_*$ is satisfiable, and if $(I_*, T_*)$ satisfies (6.3), then $\mathcal{C}$ is consistent and we can consider the component

$$
M_{\mathcal{C}} = \text{def} \ (V^{\text{in}}, V^{\text{out}}, I_*, T_*).
$$

The following theorem holds, which justifies the above notations:

**Theorem 6.2.**

1. When seeing $\mathcal{C}$ as an A/G contract, $E_{\mathcal{C}}$ is the maximal environment for $\mathcal{C}$, and $M_{\mathcal{C}}$ is the maximal implementation of $\mathcal{C}$, see (5.5).

2. The map $(T_A, T_A \Rightarrow T_G) \mapsto M_{\mathcal{C}}$ is nondecreasing, when predicates are equipped with the order inherited from $\mathcal{r} \leq \mathcal{t}$ and components are ordered by inclusion. □
Proof. Statement 1 holds by the very definition of the Moore game. We thus focus on Statement 2. To prove it, it is enough to prove by induction that

\[ \text{the map } (T_A, T_A \Rightarrow T_G) \mapsto (C_k, T_{k+1}, T_A \Rightarrow C'_k) \text{ is nondecreasing} \quad (6.10) \]

Property (6.10) holds for \( k = 0 \) by construction, since \( C_0 = \tau \) and \( T_1 = \forall (V^{in})', [T_A \Rightarrow T_G] \). Assume that (6.10) holds until \( k - 1 \) and consider two pairs \((T_{A_1}, T_{G_1})\) and \((T_{A_2}, T_{G_2})\) such that

\[ T_{A_1} \leq T_{A_2} \text{ and } [T_{A_1} \Rightarrow T_{G_1}] \leq [T_{A_2} \Rightarrow T_{G_2}] \]

By the induction assumption we have

\[ C^{1}_{k-1} \leq C^{2}_{k-1} \text{ and } T^1_k \leq T^2_k \text{ and } [T_{A_1} \Rightarrow C^{1'}_k] \leq [T_{A_2} \Rightarrow C^{2'}_k] \]

Using (6.6) we get, on the one hand,

\[ C^1_k = C^1_{k-1} \wedge \exists (V^{out})'.T^1_k \leq C^2_{k-1} \wedge \exists (V^{out})'.T^2_k = C^2_k \]

which implies, since \( T_{A_1} \leq T_{A_2} \)

\[ [T^1_A \Rightarrow C^{1'}_k] \leq [T^2_A \Rightarrow C^{2'}_k] \]

On the other hand, we have:

\[ T^1_{k+1} = \forall (V^{in})'.[T^1_A \Rightarrow (T^1_G \wedge C^{1'}_k)] \]

\[ = \forall (V^{in})'.[(T^1_A \Rightarrow T^1_G) \wedge (T^1_A \Rightarrow C^{1'}_k)] \]

\[ \leq \forall (V^{in})'.[(T^2_A \Rightarrow T^2_G) \wedge (T^2_A \Rightarrow C^{2'}_k)] \]

\[ \leq T^2_{k+1} \]

which finishes the proof of Statement 2. \( \Box \)

6.4 Moore Interfaces, seen as A/G contracts

6.4.1 Parallel Composition

We continue our development of the link between Moore Interfaces and A/G contracts by considering the parallel composition. The parallel composition and associated compatibility property were the motivation for the authors
6.4. Moore Interfaces, seen as A/G contracts

Two Moore Interfaces \( C_1 \) and \( C_2 \) are composable if \( V_{\text{out}}^{1} \cap V_{\text{out}}^{2} = \emptyset \) and their parallel composition should then coincide with the composition \( C_1 \otimes C_2 \) where \( C_1 \) and \( C_2 \) are seen as A/G contracts.

Returning to A/G contracts, if \( C_1 \) and \( C_2 \) are two A/G contracts in saturated form, then we have seen that their parallel composition is given by the assume/guarantee pair

\[
C_1 \otimes C_2 = ([A_1 \cap A_2] \cup \neg[G_1 \cap G_2], G_1 \cap G_2).
\]

(6.11)

We immediately see that the computation of this parallel composition can be performed as follows:

1. Introduce the dual contract \( \overline{C} = (G_1 \cap G_2, A_1 \cap A_2) \);
2. Compute its saturated form \( (G_1 \cap G_2, [A_1 \cap A_2] \cup \neg[G_1 \cap G_2]) \);
3. Take the dual of the result.

The key point is that step 2 can be performed by computing the winning strategy of the game associated to \( \overline{C} \), seen as a Moore Interface. This indeed yields the algorithm originally presented in equation (1) of [82] for checking compatibility:

\[
T_{k+1} = \forall (V_{\text{out}}'). (T_{G_1} \wedge T_{G_2}) \Rightarrow (T_{A_1} \wedge T_{A_2} \wedge C_k')
\]

\[
C_{k+1} = C_k \wedge \exists (V_{\text{in}}'). T_{k+1}
\]

(6.12)

This is summarized in the following result:

**Theorem 6.3.** Computing the parallel composition of two saturated contracts \( C_1 \otimes C_2 \), as defined in (6.11), is achieved by computing the fixpoint of the algorithm originally presented in equation (1) of [82] for checking compatibility.

6.4.2 Refinement

We now compare the refinement relation \( C_2 \preceq C_1 \) stated in Definition 5.4 for saturated contracts, with the alternating simulation of the game \( \Gamma_{C_2} \) by the game \( \Gamma_{C_1} \), as proposed in [82]. The quotes from [82] on page 205 suggests
that this alternating refinement should coincide with the refinement for A/G contracts. We now investigate this question.

Let \( G_i = (V_i^{\text{in}} \cup V_i^{\text{out}}, I_A, I_G, T_A, T_G), i = 1, 2, \) be two Moore Interfaces and denote by \( (A_i, G_i) \) their associated A/G contracts. Following Definition 5.4 of Section 5, we have

\[
(A_2, G_2) \preceq (A_1, G_1) \quad \text{iff} \quad \left\{ \begin{array}{l}
E_{\phi_2} \supseteq E_{\phi_1} \quad (a) \\
M_{\phi_2} \subseteq M_{\phi_1} \quad (b)
\end{array} \right.
\] (6.13)

By Statement 2 of Theorem 6.2, a sufficient condition for the right hand side of (6.13) to hold is

\[
\left\{ \begin{array}{l}
I_{A_1} \Rightarrow I_{A_2} \quad \text{and} \\
I_{G_2} \Rightarrow I_{G_1}
\end{array} \right. \quad (a)
\left\{ \begin{array}{l}
T_{A_1} \Rightarrow T_{A_2} \quad \text{and} \\
T_{G_2} \Rightarrow T_{G_1}
\end{array} \right. \quad (b)
\] (6.14)

Following Definition 5 of [82] with appropriate change of notations and taking into account the fact that the alphabet of actions \( V \) is fixed, we have

\[
\Gamma_{\phi_2} \preceq \Gamma_{\phi_1} \quad \text{(in the sense of the alternating simulation)} \quad \text{iff} \quad V_{\text{in}}^2 = V_{\text{in}}^1 \quad \text{and} \quad \text{the following formulas are valid:}
\]

\[
\left\{ \begin{array}{l}
I_{A_1} \land I_{G_2} \Rightarrow I_{A_2} \land I_{G_1}
\end{array} \right. \quad \text{and} \quad \left\{ \begin{array}{l}
T_{A_1} \land T_{G_2} \Rightarrow T_{A_2} \land T_{G_1}
\end{array} \right.
\] (6.15)

Setting \( Q = I_A \) or \( T_A \) and \( P = I_G \) or \( T_G \), we wish to check the following:

\[
[Q_2 \Rightarrow P_2] \Rightarrow [Q_1 \Rightarrow P_1] \iff [Q_1 \land P_2 \Rightarrow Q_2 \land P_1]
\]

On the one hand we have:

\[
[Q_1 \land P_2 \Rightarrow Q_2 \land P_1] = [Q_2 \land P_1] \lor \neg [Q_1 \land P_2] = [Q_2 \land P_1] \lor \neg Q_1 \lor \neg P_2 = [Q_2 \lor \neg Q_1 \lor \neg P_2] \land [P_1 \lor \neg Q_1 \lor \neg P_2]
\]

On the other hand, we have:

\[
[Q_2 \Rightarrow P_2] \Rightarrow [Q_1 \Rightarrow P_1] = [P_2 \lor \neg Q_2] \Rightarrow [P_1 \lor \neg Q_1] = P_1 \lor \neg Q_1 \lor \neg P_2 \land Q_2 = [P_1 \lor \neg Q_1 \lor \neg P_2] \land [P_1 \lor \neg Q_1 \lor Q_2] = [Q_2 \lor \neg Q_1 \lor P_1] \land [P_1 \lor \neg Q_1 \lor \neg P_2]
\]

The two expressions differ by the two terms in red. Now, take (6.14-a) into account, i.e., \( Q_1 \Rightarrow Q_2 \), we get that the substitution \( P_1 \leftrightarrow \neg P_2 \) is absorbed
by the tautology $Q_2 \lor \neg Q_1$. Thus,

assuming (6.14-a), then (6.14-b) and (6.15) are equivalent. \hspace{1cm} (6.16)

Hence, we can state:

**Theorem 6.4.** Augmenting the alternating refinement (6.15) with condition (6.14-a) makes it stronger than A/G contract refinement. \hspace{1cm} □

The possible gap between alternating refinement and A/G contract refinement lies in the fact that (6.14) is only sufficient for A/G contract refinement. Having (6.14) *restricted to the set of reachable states* is necessary and sufficient.

The bottom line is that the refinement developed in [82] seems to ignore the condition regarding assumptions. Interestingly enough, the authors were able to relate refinement to parallel composition as expected: parallel composition is monotonic w.r.t. refinement, thus supporting independent development. The following question arises then:

Is there really any added value in paying attention to both implementations and environments as we did in A/G contracts (and in the meta-theory)?

So, what are we missing for sure if we do not handle environments as first class citizens? The answer lies in the meta-theory. One property is lost by Moore Interfaces à la Chakrabarti, namely:

If $E$ is a legal environment for the composition $\mathcal{C}_1 \otimes \mathcal{C}_2$, and $M_1$ is an implementation of $\mathcal{C}_1$, then $E \times M_1$ is a legal environment for $\mathcal{C}_2$.

This is a missing property in Moore Interfaces — even in the mind of the authors, see the quotes from [82] reproduced on page 205 — and we believe its lack weakens somehow Moore Interfaces as a support for independent development.

### 6.5 Discussion

One can say that our contribution in this chapter is to mildly modify the Moore Interfaces to make them equivalent to A/G contracts and thus meta-theory compliant, with the advantage of being computationally effective.
We think that the term “interface” used by the authors of [82] is in disagreement with our terminology — we nevertheless kept this term for our exposure. Indeed the “synchronous interfaces” are not an interface model, in which environments and implementations are folded into a single entity: the “interface”. In Moore Interfaces, we rather have two entities $T_A$ and $T_G$, although both act on the same underlying set of variables. The tight link between Moore Interfaces and A/G contracts — they are nearly identical — that we have just established, further justifies this standpoint. We believe that this link is beneficial both for the A/G contracts and the Moore Interfaces. For A/G contracts, it provides a solution to the embarrassing issue of contract saturation. For Moore Interfaces it points out a (seemingly) missing condition in the alternating refinement.

Reference [82] also generalizes the Moore Interfaces to Bidirectional Interfaces. Bidirectional Interfaces offer a dynamic definition of the i/o profile and initial and transition predicates, in that the decomposition $V = V^{\text{in}}(q) \cup V^{\text{out}}(q)$ and predicates $I_A(q), I_G(q)$ and $T_A(q), T_G(q)$ depend on some location $q \in Q$, where the location $q$ evolves according to a deterministic transition system whose transitions are guarded by predicates over the variables of $V$. This additional flexibility preserves the possibility of considering the game $\Gamma_\epsilon$. The follow-up paper [118] studies the conjunction of such interfaces, under the term of shared refinement.

As a final observation, Moore Interfaces require finite domains for their variables. Clearly, contract frameworks allowing for any type of data are needed. By only manipulating abstract assertions (sets of behaviors), A/G contracts offer this possibility. In this case, of course, the contract algebra is no longer effective, hence, in Chapter 4 we proposed semi-decision procedures based either on observers (a kind of test) or on abstractions. It may be worth exploring how to extend the Moore Interfaces to this situation. Can Moore Games still be defined? Can we propose semi-decision procedures based on Moore Games? Is this any superior to the existing approaches?
Rely/Guarantee reasoning was proposed in the community of “formal methods” as a way to reason about imperative programs with shared variable concurrency [159, 160, 91, 161]. It is one of the important bases of the VDM method [163]. In this section we investigate the connections between Rely/Guarantee reasoning and A/G contracts.

### 7.1 A brief on rely/guarantee reasoning

Figure 7.1 shows the syntax of the mini-language considered in [91], for shared variable concurrency — we refer to it as $\mathcal{L}$ throughout this chapter.

\[
\begin{align*}
\text{Stmt} & \ = \ Par \mid While \mid If \mid Seq \mid Assign \mid skip \\
\text{Par} & \ ::= \ [sl : \text{Stmt}] \ [sr : \text{Stmt}] \\
\text{While} & \ ::= \ [b : \text{Expr}] \ [body : \text{Stmt}] \\
\text{If} & \ ::= \ [b : \text{Expr}] \ [body : \text{Stmt}] \\
\text{Seq} & \ ::= \ [sl : \text{Stmt}] \ [sr : \text{Stmt}] \\
\text{Assign} & \ ::= \ [id : \text{Id}] \ [e : \text{Expr}] \\
\text{Expr} & \ = \ D \mid Id \mid \text{Dyad} \\
\text{Dyad} & \ ::= \ [op] \ [a : \text{Expr}] \ [b : \text{Expr}]
\end{align*}
\]

**Figure 7.1:** Abstract syntax of the language considered in [91] — we call it $\mathcal{L}$. We do not detail what the operators $op$ are. Data domains are assumed finite and equal to $D$ and we assume that the Boolean domain $\mathbb{B}$ is contained in $D$. 
The semantics of the underlying imperative sequential language (obtained by removing the Par operator of Figure 7.2) is clear. The only important point about the parallel operator Par is that it operates by interleaving actions from sl and sr. To highlight that assignments are used to update variables in imperative programs, we call them state variables in the sequel and denote by X the underlying set of state variables.

Figure 7.2-top shows an example of concurrent program [161]. The program begins by an assignment and then launches two concurrent programs (threads). The two programs interact via their shared variables v, r, b, f where f is a flag taking the two values “wr” (write) and “rd” (read). This L program is complemented by associated rely and guarantee conditions, for the left and right threads, respectively. The discussion of rely and guarantee conditions is deferred to Section 7.3.2 and Definition 7.5.

In L, parallel compositions can be nested with any other constructs. To simplify our discussion, we restrict ourselves to programs that have been flattened, i.e. that are n-ary parallel compositions of purely sequential imperative programs:

\[ P = \parallel_{i=1} P_i \], where \( P_i \) is sequential.

The semantics \( [[P]] \) of an L program \( P \) is defined as being the set of its traces, which are the finite or infinite sequences

\[ \tau : \mathbb{N} \rightarrow X \times D \]
7.2. Components for shared variable concurrency

of successive assignments of a value to one of the state variables, that can be produced as the result of executing \( P \). This definition of the traces pinpoints the interleaving nature of the semantics.

For our development, we follow the methodology recommended by the meta-theory of Chapter 4. We first start by defining what the class of components is. Then, we introduce contracts.

### 7.2 Components for shared variable concurrency

Our approach to define components is simple:

1. We recall how the class of imperative programs with shared variable concurrency (i.e., the \( L \) language) can be mapped to the class of synchronous transitions systems. This is more or less folklore and known, but we are not aware of any comprehensive exposure of it. We therefore give a full development.

2. We show that the image of \( L \) by this mapping is closed under the synchronous composition of transition systems. We can thus consider this image of \( L \) as our class of components.

3. Since this class of components is just a subclass of the synchronous assertions considered in Section 5.1, we can simply use A/G contracts for it.

Our approach is illustrated on Figure 7.3.

![Figure 7.3: Getting A/G Contracts for imperative programs with shared variable concurrency](image)

In the next section we develop the mapping.
7.2.1 From concurrent programs to synchronous assertions

We now map $\mathcal{L}$ to transition systems as follows. We assume an additional underlying set $\mathcal{V}$ of variables, whose domain is $D \cup \{\bot\}$, where the extra value $\bot$ models absence, see Section 5.1 and we equip $D \cup \{\bot\}$ with the flat order by which $\bot$ is less than any element of $D$:

$$\forall d \in D : \bot < d$$

In Section 5.1 we defined stuttering invariance for a transition system possessing only variables. We now extend this notion to transition systems possessing both state variables and variables: $X \cup \mathcal{V}$. In words, the stretching operator introduced in (5.2) is defined as sustaining the most recent value of a state variable in case of stuttering. Formally, we redefine the silent reaction $\varepsilon$ as follows:

$$(7.1)$$

That is, a silent reaction takes as an input a valuation for all the state variables (belonging to $X$) and assigns the absent value $\bot$ to all variables (belonging to $\mathcal{V}$). The stretching operator introduced in (5.2) is redefined as follows:

$$(7.2)$$

By replacing (5.2) by (7.2) we extend stuttering invariance to systems having both variables and state variables. Say that a behavior $\sigma$ is stuttering free if it contains no silent reaction.

We are now ready to define the mapping, from concurrent programs to synchronous assertions. In the following, for $v \in \mathcal{V}$, the expression

$$\text{present}(v)$$

defines an event, equal to $\tau$ when $v \neq \bot$ and otherwise equal to $\bot$. We next assume an index set $I$ that will serve us to uniquely identify sequential programs from $\mathcal{L}$. To each variable $x \in X$ we associate the set $\{x_i \in \mathcal{V} | i \in I\}$ of
variables and the stuttering invariant transition system $M_x$ defined as follows, where $i$ and $i'$ range over the index set $I$:

$$ M_x : \begin{cases} \forall i \neq i' \Rightarrow \text{present}(x_i) \land \text{present}(x_{i'}) = \bot \\ x = \begin{cases} \exists i. \text{present}(x_i) \text{ then } x_i \text{ else } \text{pre}(x) \end{cases} \end{cases} \quad (7.3) $$

The first equation states that $x_i$ and $x_{i'}$ are never present in the same reaction if $i \neq j$ (the infimum refers to the flat order on $D \cup \{\bot\}$). The second equation states that $x$ is updated by taking the value of the (at most unique) present $x_i$ and is otherwise unchanged ($\text{pre}(x)$ returns the value of $x$ at the previous reaction). Then we define the *interleaving system* $M$ defined as the following parallel composition:

$$ M = \text{def } \prod_{x \in X} M_x \quad (7.4) $$

Composing any assertion $N$ with $M$ keeps only the behaviors of $N$ in which updates of different variables interleave.

**Definition 7.1 (mapping $P \mapsto \hat{M}(P)$).** Let $J \subseteq I$ and $P = \|_{j \in J} P_j$ be a program from $\mathcal{L}$, consisting of a parallel composition of the sequential programs $P_j$. We map $P$ to the synchronous transition system $\hat{M}(P)$ defined as follows:

$$ \hat{M}(P) = \text{def } M \times \prod_{j \in J} \hat{P}_j \quad (7.5) $$

where $\hat{P}_j$ is the stuttering invariant system obtained, from $P_j$ by:

1. Mapping every assignment $x \leftarrow \text{Expr}$ in $P_j$ to the transition system $x_j = \text{Expr}$ where $j$ ranges over $J$, and

2. Making the resulting system stuttering invariant by including the set of all its stretched behaviors following (7.2). \qed

We will need the following auxiliary assertion, where $J \subseteq I$:

$$ M_J = \text{def } \prod_{x \in X} \left( M_x \times \prod_{i \in J} \text{present}(x_i) = \bot \right) \quad (7.6) $$

Assertion $M_J$ states that every sequential process not belonging to $J$ is silent. The following lemma expresses that the mapping $P \mapsto \hat{M}(P)$ is faithful:

**Lemma 7.1.** The mapping $P \mapsto \hat{M}(P)$ satisfies the following properties:
1. \( P = ||_{j \in J} P_j \) can be uniquely recovered from \( \hat{M}(P) \) as follows:

   (a) set \( \hat{M}(P) \equiv_M M_J \times \hat{M}(P) \): by doing so we keep only those behaviors of \( \hat{M}(P) \) in which sequential processes not belonging to \( J \) keep silent;

   (b) keep only in \( \hat{M}(P) \) the behaviors that are stuttering free; by construction, the remaining behaviors of \( \hat{M}(P) \) are such that, at any reaction, exactly one variable \( x_j \), for \( x \in X, j \in J \), is present whereas all other variables are absent; let \( d \) be the value carried by the present \( x_j \) and let \( x \) be the corresponding state variable;

   (c) return the trace of \( P \) consisting of the successive pairs \( \{ \text{state variable } x, \text{value } d \} \).

2. We have \( \hat{M}(P_1 \parallel P_2) = \hat{M}(P_1) \times \hat{M}(P_2) \), where \( \times \) is the composition of synchronous assertions defined in (5.3).

\[ \hat{M}(P_1 \parallel P_2) = M \times \prod_{j \in J_1 \cup J_2} \hat{P}_j = M \times \prod_{j \in J_1} \hat{P}_j \times \prod_{j \in J_2} \hat{P}_j \]

\[ = \left( M \times \prod_{j \in J_1} \hat{P}_j \right) \times \left( M \times \prod_{j \in J_2} \hat{P}_j \right) = \hat{M}(P_1) \times \hat{M}(P_2) \]

(The duplication of \( M \) in the product is legitimate because \( M \times M = M \).)

\[ \square \]

7.2.2 From predicates on \( L \) to synchronous predicates

It is of interest to see how state transition predicates \( b : D^X \times D^X \rightarrow \mathbb{B} \) are transformed under the mapping \( P \mapsto \hat{M}(P) \).

**Lemma 7.2.** Let \( P = ||_{i \in I} P_i \) be as in Definition 7.1, and, for each \( i \in I \), let \( b_i \) be a state transition predicate satisfied by \( P_i \). Then \( P \) satisfies the disjunction \( \bigvee_{i \in I} b_i \).

\[ \square \]
Proof. Immediate, resulting from the fact that the composition is by shared variable and interleaving: one and only one $P_i$ updates some variables at a given time. □

In the mapping $P \mapsto \bar{M}(P)$, state variables sitting on the left hand side of an assignment are mapped to $I$-indexed variables belonging to $\mathcal{V}$, while other instances of state variables remain unchanged, see Definition 7.1. Should we lift predicates on $P$ to predicates on $\bar{M}(P)$ based on the above mapping of state variables to variables? Not quite. What we aim at instead is to evaluate a predicate exactly when its variables are present, and to keep it neutral otherwise. This leads to the following definition:

Definition 7.2. Let $P = \|_{j \in J} P_j$ be as in Definition 7.1. Let $j \in J$ and $b_j$ be some predicate on the state variables of $P_j$. We map $b_j$ to the following predicate $\hat{b}_j$ on $\bar{M}(P)$:

$$\hat{b}_j \overset{\text{def}}{=} \begin{cases} \text{present}(j) \text{ then } b_j \text{ else } \top \\ \end{cases}$$

where $\text{present}(j) \overset{\text{def}}{=} \bigvee_{x \in X} \text{present}(x_j)$ (7.7)

is the event that characterizes the set of reactions at which $\hat{P}_j$ is updating at least one variable, i.e., is not stuttering. □

Observe that $\hat{b}_j$ defined in (7.7) is a state transition predicate, i.e.:

$$\hat{b}_j : D^X \times D^X \to \mathbb{B}$$

Lemma 7.3. Let $P = \|_{j \in J} P_j$ be as in Definition 7.1, and, for each $j \in J$, let $b_j$ be a state transition predicate satisfied by $P_j$. Then $\bar{M}(P)$ satisfies the conjunction $\bigwedge_{j \in J} \hat{b}_j$.

Proof. First, observe that $P_j$ satisfies $b_j$ if and only if $\hat{P}_j$ satisfies $\hat{b}_j$. The conclusion follows from the fact that $\bar{M}(P) = \prod_j \bar{M}(P_j)$ and the fact that the composition $\times$ is by intersection. □

Lemmas 7.2 and 7.3 read as follows: through the mapping $P \mapsto \bar{M}(P)$, the disjunction of predicates attached to different sequential programs maps to the conjunction thereof.
7.2.3 Defining components

We are now ready to define our class of components:

**Definition 7.3.** A *shared variable component* is any synchronous assertion that is the image of a program of $L$ under mapping $\mathcal{M}()$. □

By Lemma 7.1 the so defined subset of assertions is closed under composition $\times$. Hence, our class of components satisfies the conditions required by the meta-theory of Chapter 4. On the other hand, observe that, if $M$ is a component in the sense of Definition 7.3, its complement $\neg M$ is generally not, since the interleaving condition is not stable under complement.

7.3 Contracts for shared variable concurrency

In this section we introduce contracts and relate them, in part, to Rely/Guarantee reasoning.

7.3.1 Shared Variable contracts

Besides the different naming, the following is indeed a verbatim of Definition 5.3:

**Definition 7.4.** A *shared variable contract* is a pair $C = (A, G)$ of assertions, called the *assumptions* and the *guarantees*. The set $E_C$ of the legal environments for $C$ collects all components $E$ such that $E \subseteq A$. The set $M_C$ of all components implementing $C$ is defined by $A \times M \subseteq G$. □

Being just $A/G$ contracts, the shared variable contracts inherit their algebra for reasoning. Due to Definition 7.3, however, the associated class of components is different from the one stated in Definition 5.2 of Section 5.1. In particular, no simple formula exists to characterize the maximal environment or implementation. We not even know there is a unique one. This makes it more difficult to develop a dialect of $A/G$ contracts for shared variable concurrency. This may also explain why the algebra of rely/guarantee reasoning seems complex and the different developments of it exhibit variations [159, 163, 160, 91, 161].
7.3. **Contracts for shared variable concurrency**

### 7.3.2 A comparison with Rely/Guarantee reasoning

We shall nevertheless try to compare the proof rule of rely/guarantee reasoning given in [160] for the parallel composition of shared variable imperative programs, with the composition of A/G contracts. The following proof rule for Rely/Guarantee conditions is stated in [162], for the parallel composition, where superscript (*) denotes the transitive closure:

\[
\{p, r \lor g_2\} P_1 \{g_1, q_1\} \\
\{p, r \lor g_1\} P_2 \{g_2, q_2\} \\
\{p, r\} P_1 \parallel P_2 \{g_1 \lor g_2, q_1 \land q_2 \land (r \lor g_1 \lor g_2)^*\}
\]

(7.8)

The intuition of this rule is the following (verbatim from [162]): If the overall combination of statements composing \( P_1 \parallel P_2 \) has to be able to achieve its post condition with interference \( r \) from its environment, then each \( P_i \) has to be able to tolerate that degree of interference plus any that can come from the sibling process \( P_j \); the overall guarantee condition is the disjunction of the guarantees of the components; the overall post condition is at least as strong as the conjunction of the post conditions of the components but it is possible to add a conjunct that is the reflexive closure of the guarantees and the overall rely condition. A “precis of rely/guarantee reasoning” is available in the Appendix of [162].

Rely/Guarantee reasoning is revisited now in the context of our work on contracts.

**Definition 7.5.** For \( P \) a \( \mathcal{L} \) program, and predicates \( p : D^X \rightarrow \mathbb{B} \) and \( r, g, q : D^X \times D^X \rightarrow \mathbb{B} \), statement \( \{p, r\} P \{g, q\} \) means that,

when program \( P \) operates

on a state satisfying the **precondition** \( p \) on the initial states,

under an environment satisfying the **rely** \( r \) on the transitions,

then

the resulting new state will satisfy the **postcondition** \( q \) on the initial and final states, and

the effect on the environment will satisfy the **guarantee** \( g \) on the transitions.
An example of Rely/Guarantee conditions in a parallel construct is given in Figure 7.2-bottom. These conditions state the rely and guarantee conditions for the producer and consumer, respectively. The matching between rely and guarantee conditions, for the producer and the consumer, indicates that the union of all the guarantees will hold for their parallel composition, with the trivial ($= \tau$) overall rely condition. This is used by the authors of [161] to prove that the program of Figure 7.2-top indeed meets the specification of a one-place buffer. The following is a verbatim from [161]:

Testing and setting flag $f$ in Figure 7.2 ensures that the producer and consumer alternate their access to $b$. During its read phase, the consumer needs to rely on the fact that the value of $b$ cannot change but this is too strong as a rely condition for the whole of the consumer process — the producer process could never insert anything into the buffer if it were required to achieve a guarantee condition of $b'=b$. But the consumer process can instead rely on $f=rd \Rightarrow b'=b$, which in turn is easy for the producer to guarantee. The “monotonic” behavior of the flags means that the producer has also to guarantee that $f=rd \Rightarrow f'=rd$ and the consumer must guarantee $f=wr \Rightarrow f'=wr$.

We now use the Sections 7.2.1 and 7.2.2 to derive how rule (7.8) is transformed under the map $M()$. Let $\hat{p}, \hat{r}, \hat{g}_1, \hat{g}_2, \hat{g}$ be the images, under the map $M()$, of the predicates $p, r, g_1, g_2, g$.

Let $A$ be the assertion generated by the state predicate $\hat{p}$ as initial condition, and $\hat{r}$ as transition relation. The symmetric definition holds for $A_2$. Let $G_1$ be the conjunction of the predicate $\hat{q}_1$ on the initial and final states, and the assertion generated by the transition relation $\hat{g}_1$. The corresponding definition holds for $G_2$. Finally, let $G$ be the conjunction of the predicate $\hat{q}$ on the initial and final states, and the assertion generated by the transition relation $\hat{g}$. This defines the contracts $\mathcal{C} = (A, G)$ and $\mathcal{C}_i = (A_i, G_i)$ for $i = 1, 2$.

Rule (7.8) rewrites as follows, where $M_i =_{\text{def}} M(P_i)$:

$$M_1 \models^u \mathcal{C}_1 = (A \cap G_2, G_1) \quad \text{and} \quad M_2 \models^u \mathcal{C}_2 = (A \cap G_1, G_2)$$

$$M_1 \times M_2 \models^u \mathcal{C} = (A, G_1 \cap G_2) \quad (7.9)$$

To simplify the comparison, we will focus on the interactions with the environment and ignore the pre- and postconditions by setting $p = q = q_1 = q_2 = \tau$, so that the special condition on the pre- and postconditions in (7.9) is vacuously satisfied. Comparing (7.9) with the definition of the A/G contract
composition (see (5.8,5.9) and (4.1,4.2)) shows that taking $G = G_1 \cap G_2$ in (7.9) yields $C = C_1 \otimes C_2$, the parallel composition of the two contracts.

7.4 Discussion

The above analysis strongly suggests that close ties exist between A/G contracts and Rely/Guarantee reasoning used in the VDM method. Further studies would be required to investigate these ties more closely.

Rely/Guarantee rules, however, have been developed as a proof system for the VDM specification method, not as a contract framework. For example, the “precis of rely/guarantee reasoning” given in [162] associates a rule to each particular feature of the underlying language (which is a variation of $L$). This makes the resulting set of rules more complex and less abstract than our A/G contract framework. As a consequence, a comprehensive comparison would be more difficult. Another difficulty arises from the consideration, in Rely/Guarantee reasoning, of pre- and postconditions as first class citizens, whereas they are implicitly handled as part of assumptions and guarantees in A/G contracts.

Our approach suggests a way to equip the $L$ language with A/G contracts: apply the map $P \mapsto M(P)$ and then use standard A/G contracts. Unfortunately, though the map $M()$ is conceptually simple, it is not practical as it requires considering in advance the set of all possible parallel threads (for indexing by $I$). The adequate approach probably consists in mapping backward A/G contracts to $L$ by reverting the map $M()$ in some appropriate way. This remains to be done.
Interface theories are an interesting alternative to Assume/Guarantee contracts. They aim at providing a merged specification of the implementations and environments associated to a contract via the description of a single entity, called an interface. We review some typical instances of interface theories, with emphasis on Interface Automata and Modal Interfaces.

We restrict ourselves to deterministic interfaces, in that a given action can trigger at most one transition at a given state. This restriction is acceptable for specifications, which are anyway meant to be abstract. Non-deterministic interfaces have been studied (see the bibliographical section). They are way more complex, both in the concepts and in computational cost for their associated algorithms.

Interface theories generally use (a mild variation of) Lynch Input/Output Automata [194, 193] as their framework for components and environments. As a prerequisite, we thus recall the background on Input/Output Automata, i/o-automata for short. Again, we restrict ourselves to deterministic i/o-automata. This restriction is only technical for the sake of simplicity, not essential, however. The cost of handling non-determinism in components is only notational.
8.1 Components as i/o-automata

For our notion of component, we follow closely the motivations developed in [105]. The aim is that components possess input and output actions and are composed via input-to-output connections. Unlike contracts, seen as specifications, components cannot refuse actions from the environment; formally, components are receptive.

8.1.1 Input/Output Automata

Definition 8.1 (i/o-automaton). An i/o-automaton is a tuple

\[ M = (\Sigma^\text{in}, \Sigma^\text{out}, Q, q_0, \rightarrow), \]

where \( \Sigma^\text{in}, \Sigma^\text{out} \) are disjoint finite input and output alphabets; \( Q \) is a finite set of states and \( q_0 \in Q \) is the initial state; \( \rightarrow \subseteq Q \times \Sigma \times Q \) is the transition relation, where \( \Sigma = \Sigma^\text{in} \cup \Sigma^\text{out} \). □

As usual, we write \( q \xrightarrow{\alpha} q' \) to mean \((q, \alpha, q') \in \rightarrow\) and \( q \xrightarrow{\alpha} \) to indicate the existence of a \( q' \) such that \( q \xrightarrow{\alpha} q' \). An i/o-automaton can be interpreted as an open system: the transitions labeled by actions in \( \Sigma^\text{out} \) represents the outputs that the system can generate while the transitions labeled by actions in \( \Sigma^\text{in} \) represent the inputs a system can accept. By concatenation, the transition relation \( \rightarrow \) extends to a relation \( \rightarrow^* \) on \( Q \times \Sigma^* \times Q \), where \( \Sigma^* \) is the set of all finite words over \( \Sigma \). Say that a state \( q' \) is reachable from \( q \) if there exists some word \( w \) such that \( q \xrightarrow{w} q' \). To simplify our development we restrict ourselves to deterministic i/o-automata, i.e.:

\[ [q \xrightarrow{\alpha} q_1 \text{ and } q \xrightarrow{\alpha} q_2] \implies q_2 = q_1 \quad (8.1) \]

and we denote by

\[ \alpha \mapsto \delta(q, \alpha) \quad (8.2) \]

the partial function such that \( \delta(q, \alpha) \) is the unique (if it exists) state such that \( q \xrightarrow{\alpha} \delta(q, \alpha) \). Two i/o-automata \( M_1 \) and \( M_2 \) having identical alphabet \( \Sigma \) are composable if the usual input/output matching condition holds: \( \Sigma_1^\text{out} \cap \Sigma_2^\text{out} = \emptyset \).
and their composition $M = M_1 \times M_2$ is given by

$$
\begin{align*}
\Sigma^{\text{out}} &= \Sigma_1^{\text{out}} \cup \Sigma_2^{\text{out}}, & \Sigma^{\text{in}} &= \left(\Sigma_1^{\text{in}} \cup \Sigma_2^{\text{in}}\right) \setminus \Sigma^{\text{out}} \\
Q &= Q_1 \times Q_2, & q_0 &= (q_{1,0}, q_{2,0})
\end{align*}
\tag{8.3}
$$

$$(q_1, q_2) \xrightarrow{\alpha} (q_1', q_2') \quad \text{iff} \quad q_i \xrightarrow{\alpha} q_i' \text{ holds for } i = 1, 2.$$ 

For $M_i, i = 1, 2$ two i/o-automata and two states $q_i \in Q_i$, say that $q_1$ simulates $q_2$, written $q_2 \leq q_1$ if

$$
\forall \alpha, q_2' \text{ such that } q_2 \xrightarrow{\alpha} q_2' \Rightarrow \exists q_1' \text{ such that } [q_1 \xrightarrow{\alpha} q_1' \text{ and } q_2' \leq q_1'].
\tag{8.4}
$$

Say that

$M_1$ simulates $M_2$, written $M_2 \leq M_1$, if $q_{2,0} \leq q_{1,0}$. \tag{8.5}

Observe that simulation relation (8.4,8.5) does not distinguish inputs from outputs neither it distinguishes the component from its environment. It is the classical simulation relation meant for closed systems. For $M$ a component (i.e., a receptive i/o-automaton) and $q$ a reachable state of it, we consider the ready set of $M$ at $q$:

$$
\Sigma_M(q) = \{ \alpha \mid q \xrightarrow{\alpha} M \}.
\tag{8.6}
$$

The proof of the following result is immediate:

**Lemma 8.1.** For $M_1$ and $M_2$ two i/o-automata, simulation relation $q_2 \leq q_1$ rewrites as follows: $\Sigma_{M_2}(q_2) \subseteq \Sigma_{M_1}(q_1)$ holds and $\delta_{M_2}(q_2, \alpha) \leq \delta_{M_1}(q_1, \alpha)$ holds for every $\alpha \in \Sigma_{M_i}(q_2)$. \hfill \square

Variable alphabets are handled by using the mechanism of alphabet extension. For $M = (\Sigma^{\text{in}}, \Sigma^{\text{out}}, Q, q_0, \to)$ an i/o-automaton and $\Sigma' \supset \Sigma$, we define

$$
M^{\Sigma'} = (\Sigma^{\text{in}} \cup (\Sigma' \setminus \Sigma), \Sigma^{\text{out}}, Q, q_0, \to')
$$

where $\to'$ is obtained by adding, to $\to$, for each state and each added action, a self-loop at this state labeled with this action.
8.1.2 Components and Environments are receptive i/o-automata

Call receptive (or input enabled) an i/o-automaton that reacts by proper response to any input stimulus in any state:\(^1\)

\[ M \text{ is receptive iff } \forall q \in Q, \forall \alpha \in \Sigma^{in} : q \xrightarrow{\alpha}. \] (8.7)

Receptiveness is stable under parallel composition. The following simple technique can be used to make any i/o-automaton \( M \) receptive:

1. Augment \( Q \) with an extra top state \( \top \), such that \( \top \xrightarrow{\alpha} \top \) holds for every action \( \alpha \in \Sigma \);  
   \[ (8.8) \]

2. For each pair \( (q, \alpha) \in Q \times \Sigma^{in} \) such that \( \alpha \) is not enabled at \( q \), add a transition \( q \xrightarrow{\alpha} \top \).

This yields a receptive i/o-automaton that we denote by

\[ \overline{M} \] (8.9)

and we recover \( M \) from \( \overline{M} \) by removing the top state \( \top \) and all transitions leading to it. The reader may wonder why we do not only add self-loops \( \top \xrightarrow{\alpha} \top \) for any action \( \alpha \in \Sigma^{in} \), as this would be the minimal receptive extension. Our construction is motivated by its subsequent use in the forthcoming Definitions 8.3 and 8.7. Components — and consequently environments — for use in interface theories will be receptive i/o-automata.

8.2 Interface Automata with fixed alphabet

For reasons that will become clear later, we restrict the presentation of interface automata to the case of a fixed alphabet \( \Sigma \). Interface Automata are possibly non-receptive i/o-automata, whose semantics is given in terms of contracts, that is, pairs formed of a set of implementations and a set of valid environments.

\(^1\)In fact, receptiveness is assumed in the original notion of i/o-automaton by Nancy Lynch [193, 194]. We use here a relaxed version of i/o-automaton for reasons that will become clear later.
Definition 8.2 (Interface Automata [105, 8]). Call Interface Automaton a tuple

\[ \mathcal{C} = (\Sigma^\text{in}, \Sigma^\text{out}, Q, q_0, \rightarrow) \]

where \(\Sigma^\text{in}, \Sigma^\text{out}, Q\), and \(\rightarrow\) are as in i/o-automata. The initial state \(q_0\), however, may not belong to \(Q\). \(\square\)

Definition 8.3 (associated contract). If \(q_0 \notin Q\), we take the convention that \(\mathcal{E}_C = \emptyset\) (\(\mathcal{C}\) is incompatible) and \(\mathcal{M}_C = \forall\) (any component implements \(\mathcal{C}\)). If \(q_0 \in Q\) holds, \(\mathcal{C}\) defines a compatible and consistent contract by fixing a pair \((\mathcal{E}_C, \mathcal{M}_C)\) as follows:

The set \(\mathcal{E}_C\) of legal environments for \(\mathcal{C}\) collects all components \(E\) satisfying the following conditions:

1. \(\Sigma^\text{in}_E = \Sigma^\text{out}_E\) and \(\Sigma^\text{out}_E = \Sigma^\text{in}_E\). Thus, \(E\) and \(\mathcal{C}\), seen as i/o-automata, are composable;

2. For any output action \(\alpha \in \Sigma^\text{in}_E\) of environment \(E\) such that \(q \xrightarrow{\alpha}_E\) and any reachable state \((q_E, q)\) of \(E \times \mathcal{C}\), then \(q \xrightarrow{\alpha}_E\) holds.

Now define the particular environment \(E'_C \in \mathcal{E}_C\) as follows:

(a) We first consider \(E'_C = (\Sigma^\text{out}, \Sigma^\text{in}, Q, q_0, \rightarrow)\) seen as an i/o-automaton;

(b) Making \(E'_C\) receptive following (8.9) yields the desired \(E'_C\).

The set \(\mathcal{M}_C\) of the implementations of \(\mathcal{C}\) collects all components \(M\) such that i/o-automaton \(\mathcal{C}\) simulates \(E'_C \times M\) in the sense of (8.5). \(\square\)

Summary of Results 2.

1. Through Definition 8.3, the framework of Interface Automata equipped with its existing operations of refinement defined through alternating simulation, and its parallel composition, instantiates the corresponding concepts of the meta-theory.

2. The parallel composition is not associative but only sub-associative.

3. Variable alphabets are not well supported by this framework. \(\square\)
The rest of this section develops the lemmas and theorems supporting this claim. The results and techniques used here will be reused and further extended to handle Modal Interfaces in the next section. Therefore, we do not detail all the results. In particular the construction of the quotient is not developed here. We now move to justify this summary of results.

Condition 2 means that environment $E$ is only willing to emit an output if it is accepted as an input by $C$ in the composition $E \times C$. Observe that the environment $E \times C$ is such that $E \times C$ simulates $E \times C$ in the sense of (8.5) for any $E \in \mathcal{E}_E$. Moreover the construction of $E \times C$ is justified by the following lemma:

**Lemma 8.2.** The environment $E \times C$ constructed as above is maximal in $\mathcal{E}_E$ with respect to simulation relation (8.5). □

**Proof.** By construction, $E'_E \times C$ is isomorphic to $(\emptyset, \Sigma, Q, q_0, \rightarrow)$, i.e., it is obtained from $C$, seen as an i/o-automaton, by simply turning inputs to outputs. Consequently, Condition 2 holds for $E'_E$. To show the maximality of $E \times C$, we consider an arbitrary environment $E$ of $C$ and we define the following relation $\leq_E \subseteq Q_E \times (Q \cup \{\top\})$:

$$\leq_E = \{ (q_E, q) | (q_E, q) \text{ is reachable in } E \times C \} \cup (Q_E \times \{\top\})$$

We claim that $\leq_E$ is a simulation relation from $E \times C$ to $E$.

First of all the top state $\top$ trivially simulates any state from $Q_E$. It is thus enough to consider pairs $q_E \leq_E q$ such that $q \neq \top$, meaning that $(q_E, q)$ is reachable in $E \times C$. Next, let $\alpha \in \Sigma$ be such that $q_E \xrightarrow{\alpha} q$ holds. If $\alpha \in \Sigma^{\text{in}}$ then we know by Condition 2 of Definition 8.3 that $q \xrightarrow{\alpha} C$ holds, which implies $q \xrightarrow{\alpha} E \times C$ by construction of $E \times C$. If $\alpha \in \Sigma^{\text{out}}$ then $q \xrightarrow{\alpha} E \times C$ also holds since $E \times C$ is receptive (recall that $\Sigma^{\text{out}}$ is the set of input actions of the environment $E \times C$). Performing this move leads $E \times C$ either to a state belonging to $Q$, or to $\top$. Thus, performing an $\alpha$-move in both $E$ and $E \times C$ leads to a unique pair $(q'_E, q')$ that either is reachable in $E \times C$ or belongs to $Q_E \times \{\top\}$. In any case the new pair $(q'_E, q')$ again belongs to $\leq_E$. This proves (8.10) and the lemma. □

In addition to the maximal environment $E \times C$ associated to contract $C$, we will need its maximal implementation $M \times C$, constructed as follows:
We first consider \( M'_{\mathcal{C}} = (\Sigma^{\text{in}}, \Sigma^{\text{out}}, Q, q_0, \rightarrow) \), seen as an i/o-automaton;

(b) Making \( M'_{\mathcal{C}} \) receptive following (8.9) yields the component \( M_{\mathcal{C}} \).

Note that \( E_{\mathcal{C}} \times M_{\mathcal{C}} = (\emptyset, \Sigma, Q, q_0, \rightarrow) \), hence \( M_{\mathcal{C}} \) implements \( \mathcal{C} \). The following result, which justifies the name of “maximal implementation”, is proved similarly as Lemma 8.2:

**Lemma 8.3.** \( M_{\mathcal{C}} \) is maximal in \( M_{\mathcal{C}} \) w.r.t. simulation relation (8.5). \( \square \)

The following obvious result, which uses the ready sets introduced in (8.6), will be useful:

**Lemma 8.4.** Condition 2 of Definition 8.3 rewrites: for any reachable state \((q_E, q)\) of \( E \times \mathcal{C} \), the following holds: \( \Sigma^{\text{in}} \cap \Sigma_E(q_E) \subseteq \Sigma^{\text{in}} \cap \Sigma_{\mathcal{C}}(q) \). \( \square \)

The above definition of Interface Automata is heterodox, compare with the original references [105, 8]. Definition 8.3 introduces the two sets \( E_{\mathcal{C}} \) and \( M_{\mathcal{C}} \), whereas no notion of implementation or environment is formally associated to an Interface Automaton in the original definition. Also, the handling of the initial state is unusual. Failure of \( q_0 \in Q \) to hold typically arises when the set of states \( Q \) is empty. Our Definition 8.3 allows us to cast Interface Automata in the framework of the meta-theory of Table 4.1. Corresponding relations and operations must be instantiated and we do this next.

**Refinement and conjunction:** The alternating simulation is the notion of refinement originally proposed for Interface Automata [14]:

**Definition 8.4 (alternating simulation).** Let \( \mathcal{C}_i, i = 1, 2 \) be two Interface Automata. Say that two of their respective states \( q_i, i = 1, 2 \) are in *alternating simulation*, written \( q_2 \preceq q_1 \), if the following two conditions hold, where we write \( \rightarrow_i \) for short instead of \( \rightarrow_{\mathcal{C}_i} \), for \( i = 1, 2 \):

\[
\forall \alpha \in \Sigma^{\text{in}} \cap \Sigma_1 \text{ s.t. } q_1 \xrightarrow{\alpha} q'_1 \quad \Rightarrow \quad \left\{ \begin{array}{l} \alpha \in \Sigma^{\text{in}}_2, \text{ and} \\
\exists q'_2 \text{ s.t. } q_2 \xrightarrow{\alpha} q'_2 \text{ and } q'_2 \preceq q'_1 \end{array} \right. \\
\forall \alpha \in \Sigma^{\text{out}}_2 \cap \Sigma_2 \text{ s.t. } q_2 \xrightarrow{\alpha} q'_2 \quad \Rightarrow \quad \left\{ \begin{array}{l} \alpha \in \Sigma^{\text{out}}_1, \text{ and} \\
\exists q'_1 \text{ s.t. } q_1 \xrightarrow{\alpha} q'_1 \text{ and } q'_2 \preceq q'_1 \end{array} \right. \quad (8.11)
\]

Say that \( \mathcal{C}_2 \) refines \( \mathcal{C}_1 \), written \( \mathcal{C}_2 \preceq \mathcal{C}_1 \), if \( q_{2,0} \preceq q_{1,0} \). \( \square \)
Since we restrict ourselves to deterministic Interface Automata, a matching state $q'_1$ for $q'_2$ (or $q'_2$ for $q'_1$) is unique when it exists. The following result shows that alternating simulation is equivalent to contract refinement in the sense of the meta-theory:

**Theorem 8.5.** Let $C_i, i = 1, 2$ be two Interface Automata such that $q_{i,0} \in Q_i$ for $i = 1, 2$ and $\Sigma_{\text{in}}^i = \Sigma_{\text{in}}^2$, which implies $\Sigma_{\text{out}}^1 = \Sigma_{\text{out}}^2$ — we denote them by $\Sigma_{\text{in}}$ and $\Sigma_{\text{out}}$, respectively. Then,

$$ C_2 \preceq C_1 \text{ if and only if } \left\{ \begin{array}{l} E_{C_2} \supseteq E_{C_1} \\ M_{C_2} \subseteq M_{C_1} \end{array} \right. $$

The additional condition that input/output profiles should not be changed should not come as a surprise: our definition of contract associated to $C$ in Definition 8.3 calls for such a condition.

**Proof.** We begin with the “only if” part. As a prerequisite, note that, if $E \models^x C$ and $E$ simulates $E'$ in the sense of i/o-automata, then $E' \models^x C$ follows, and, if $M \models^u C$ and $M$ simulates $M'$ in the sense of i/o-automata, then $M' \models^u C$ follows. Consequently, the only if part amounts to proving

$$ q_{2,0} \leq q_{1,0} \implies \left\{ \begin{array}{l} E_{C_1} \models^x C_2 \\ M_{C_2} \models^u C_1 \end{array} \right. \quad (8.12) $$

Focus on the environments. Pick a pair $(q_{E_{C_1}}, q_2)$ that is reached in $E_{C_1} \times C_2$ by performing some word $w \in \Sigma^*$ and let $\alpha \in \Sigma_{\text{in}}$ be such that

$$ q_{E_{C_1}} \xrightarrow{\alpha} q'_{E_{C_1}}. \quad (8.13) $$

Suppose that $q_{E_{C_1}} = \top$. Let $\nu$ be the largest prefix of $w$ leading both $E_{C_1}$ and $C_1$ to a state $q'_1 \neq \top$ and $C_2$ to a state $q'_2$, and let $\alpha' \in \Sigma_{\text{out}}$ be such that

$$ q'_1 \xrightarrow{\alpha'} \top. $$

Then, by construction of $E_{C_1}$ we have $q''_1 \xrightarrow{\alpha'} q'_1$, which, since $q''_2 \leq q'_1$, implies

$$ q''_2 \xrightarrow{\alpha'} q'_2. $$

But this contradicts the assumption that pair $(q_{E_{C_1}}, q_2)$ is reached in $E_{C_1} \times C_2$ by performing $w \in \Sigma^*$. Consequently, $q_{E_{C_1}} \neq \top$. Then, by construction of
interface theories, (8.13) implies $q_{1,0} \xrightarrow{w_1} q_1$ with $q_1 = q_{E_{\mathcal{E}_1}}$, hence $q_2 \leq q_1$ follows. By Condition 2 of Definition 8.3 relative to \( \mathcal{E}_1 \):

\[
q_{E_{\mathcal{E}_1}} \xrightarrow{\alpha} q'_{E_{\mathcal{E}_1}} \quad \text{implies} \quad q_1 \xrightarrow{\alpha} q_1'
\]

which, since $q_2 \leq q_1$, implies $q_2 \xrightarrow{\alpha} q_2'$. We thus proved that (8.13) implies $q_2 \xrightarrow{\alpha} q_2'$, which proves that $E_{\mathcal{E}_1} \models^e \mathcal{E}_2$. The proof that $M_{\mathcal{E}_1} \models^u \mathcal{E}_1$ is similar.

We now move to the “if” part. From the right hand side of the statement of the theorem, we infer that

\[
E_{\mathcal{E}_2} \text{ simulates } E_{\mathcal{E}_1} \quad \text{and} \quad M_{\mathcal{E}_1} \text{ simulates } M_{\mathcal{E}_2} \quad (8.15)
\]

and we denote by $\leq^e$ and $\leq^u$ the corresponding simulation relations. We claim that the following relation $\preceq$ is an alternating simulation, from $\mathcal{E}_2$ by $\mathcal{E}_1$. For $(q_2, q_1) \in Q_2 \times Q_1$:

\[
q_2 \preceq q_1 \quad \text{iff} \quad \begin{cases} q_1 \leq^e q_2 \text{ and } \\ q_2 \leq^u q_1 \end{cases} \quad (8.16)
\]

Observe first that $q_{2,0} \leq q_{1,0}$ holds. Pick a pair satisfying $q_2 \leq q_1$, that is: $q_1 \leq^e q_2$ and $q_2 \leq^u q_1$. Let $\alpha \in \Sigma_1^{\text{out}}$ be such that

\[
\exists q'_1 \in Q_1 : q_1 \xrightarrow{\alpha} q'_1 , \quad \text{which implies} \quad \begin{cases} q_1 \xrightarrow{E_{\mathcal{E}_1}} q'_1 \\ q_1 \xrightarrow{M_{\mathcal{E}_1}} q'_1 \end{cases} \quad (8.17)
\]

by the construction of the maximal environment and implementation, respectively. Since $q_1 \leq^e q_2$ there exists $q'_2 \in Q_2 \cup \{\top\}$ such that

\[
q_2 \xrightarrow{\alpha} E_{\mathcal{E}_2} q'_2 \quad \text{and} \quad q'_1 \leq^u q'_2. \quad (8.18)
\]

We cannot have $q'_2 = \top$ since only transitions by $\alpha \in \Sigma_2^{\text{out}} = \Sigma_1^{\text{out}}$ can lead to $\top$ in maximal environments. Hence, by construction of the maximal environments, we deduce

\[
q_2 \xrightarrow{\alpha} q'_2. \quad (8.19)
\]

Next, since we also have $q_2 \leq^u q_1$, we derive

\[
\exists q''_1 \in Q_1 \cup \{\top\} : q_1 \xrightarrow{M_{\mathcal{E}_1}} q''_1 \quad \text{and} \quad q'_2 \leq^u q''_1 \quad , \quad (8.20)
\]
which implies $q_i'' = q_i'$ using the right hand side of (8.17), since $M_{C_i}$ is deterministic. Combining (8.17–8.20) proves the following:

$$
\forall \alpha \in \Sigma^{in} \text{ such that } \exists q_1' \in Q_1 : q_1 \xrightarrow{\alpha} q_1' \quad \text{then} \quad \exists q_2' \in Q_2 : q_2 \xrightarrow{\alpha} q_2' \quad \text{and} \quad q_2' \preceq q_1' \quad (8.21)
$$

By using the same argument, albeit with maximal implementations instead of maximal environments and exchanging the indices 1 and 2, we obtain

$$
\forall \alpha \in \Sigma^{out} \text{ such that } \exists q_2' \in Q_2 : q_2 \xrightarrow{\alpha} q_2' \quad \text{then} \quad \exists q_1' \in Q_1 : q_1 \xrightarrow{\alpha} q_1' \quad \text{and} \quad q_2' \preceq q_1' \quad (8.22)
$$

(8.21, 8.22) express that $\preceq$ defined by (8.16) is an alternating simulation. □

Alternating simulation can be effectively checked, see [103] for issues of computational complexity. We note that we use simulation and alternating simulation and refinement relations for components and contracts, respectively. It is sufficient to use the classical simulation relation between components because we assume that components are input-enabled. In fact, for input-enabled systems, simulation and alternating simulation coincide. In addition, for deterministic (contracts) components, (alternating) simulation also coincides with (alternating) language inclusion. Unfortunately, no simple formula for the conjunction of contracts is known. See [118] for results in this direction, and particularly the important results by Bujtor and Vogler [64, 66] in which mistakes of previous frameworks are pointed and corrected.

**Parallel composition:** We now recall the definition of the parallel composition, for Interface Automata [103]:

**Definition 8.5.** Let $C_i, i = 1, 2$ be two Interface Automata such that $q_{i,0} \in Q_i, i = 1, 2$. The parallel composition $C_1 \otimes C_2$ is only defined if $\Sigma_1^{out} \cap \Sigma_2^{out} = \emptyset$ and, then, it is defined as follows:

1. Consider the pre-composition $C_1 \times C_2$, which is obtained by taking the product $C_1 \times C_2$ of $C_1$ and $C_2$, seen as i/o-automata, and then regarding the so-obtained product as an Interface Automaton; recall that $C_1 \times C_2 = (\Sigma^{in}, \Sigma^{out}, Q, q_0, \rightarrow)$, where the elements of this tuple are defined in (8.3);
2. Call **illegal** a state \( q = (q_1, q_2) \in Q_1 \times Q_2 \) such that
\[
\exists i, j \in \{1, 2\}, j \neq i, \exists \alpha \in \Sigma^\text{out}_i \text{ such that } q_i \xrightarrow{\alpha} \text{ but } q_j \xrightarrow{\alpha} j \quad (8.23)
\]

3. Define the subset \( E \subseteq Q \) of **exception states** as being the smallest subset of \( Q \) containing all illegal states and such that, if
\[
q \xrightarrow{\alpha} q'
\]
holds for some \( \alpha \in \Sigma^\text{out} \) and \( q' \in E \), then \( q \in E \).

4. Set \( \mathcal{C}(0) =_{\text{def}} \mathcal{C}_1 \otimes \mathcal{C}_2 \). Exception states and their incoming transitions are removed from \( \mathcal{C}(0) \), which yields an Interface Automaton \( \mathcal{C}(1) \). By doing so we may have created exception states in \( \mathcal{C}(1) \). So we remove them as well. Doing this repeatedly yields a sequence \( \mathcal{C}(k), k = 0, 1, \ldots \) such that \( Q(0) \supseteq Q(1) \supseteq \ldots \) so that this procedure converges to a fixpoint \( \mathcal{C}(K) \) in finitely many steps (\( K \) is thus finite). We define \( \mathcal{C}_1 \otimes \mathcal{C}_2 \) as being this fixpoint \( \mathcal{C}(K) \).

Note that \( \mathcal{C}_1 \otimes \mathcal{C}_2 \) is both consistent and compatible if and only if \( Q(K) \neq \emptyset \).

The relation with the meta-theory is established by the following theorem:

**Theorem 8.6.** \( \mathcal{C}_1 \otimes \mathcal{C}_2 \) as defined in Definition 8.5 instantiates the parallel composition of the meta-theory.

**Proof.** To make the proof easier to read, we restrict it to the composition of two Interface Automata.\(^2\) With reference to (4.1.4.2) where \( I = \{1, 2\} \), we prove the following
\[
\mathcal{C}_1 \otimes \mathcal{C}_2 \in \mathcal{C}_{1,2} \quad (8.24)
\]
\[
\mathcal{C}_1 \otimes \mathcal{C}_2 \text{ is minimal having this property} \quad (8.25)
\]

To prove (8.24), it is enough to consider maximal environments and implementations, i.e., to prove
\[
\begin{align*}
M_{\mathcal{E}_1} \times M_{\mathcal{E}_2} \models^\text{m} \mathcal{C}_1 \otimes \mathcal{C}_2 \quad (8.26) \\
E_{\mathcal{E}} \times M_{\mathcal{E}_2} \models^\text{e} \mathcal{C}_1 \quad (8.27)
\end{align*}
\]

\(^2\)See the subsequent discussion regarding associativity.
8.2. Interface Automata with fixed alphabet

where $\mathcal{C} = \text{def } \mathcal{C}_1 \otimes \mathcal{C}_2$. Focus first on (8.27). Let $(q_{E'}, q_{M_2}, q_1)$ be a reachable state of $E' \times M_2 \times \mathcal{C}_1$, where $\mathcal{C}_1$ is seen as an i/o-automaton, and let $\alpha \in \Sigma^\text{in}_1$ be such that

$$\alpha \in \Sigma^\text{in}_1 \implies (q_{E'}, q_{M_2}) \xrightarrow{\alpha} q_{E' \times M_2} \quad (8.28)$$

We must prove that

$$q_1 \xrightarrow{\alpha} 1 \quad (8.29)$$

follows. But this is a consequence of the following facts:

1. Due to the pruning of illegal states in constructing $\mathcal{C}$, we have $q_2 = \text{def } q_{M_2} \neq \top$ hence $q_2 \in Q_2$ and the pair $(q_1, q_2)$ is legal.

2. Since $\alpha \in \Sigma^\text{in}_1$ and since, by (8.28),

$$q_2 \xrightarrow{\alpha} q_{M_2}$$

holds, we also have $q_2 \xrightarrow{\alpha} 2$, and (8.29) follows since the pair $(q_1, q_2)$ is legal.

This proves (8.27). Focus next on (8.26). We need to prove that $\mathcal{C}$ simulates $E' \times M_{\mathcal{C}_1} \times M_{\mathcal{C}_2}$ in the sense of i/o-automata. This is proved by showing that the simulation relation $\leq (8.26)$ contains the pair of initial states, where $\leq (8.26)$ is the set of pairs $((q_{E'}, q_{M_{\mathcal{C}_1}}, q_{M_{\mathcal{C}_2}}), q)$ such that $(q_{E'}, q_{M_{\mathcal{C}_1}}, q_{M_{\mathcal{C}_2}})$ is reachable in $E' \times M_{\mathcal{C}_1} \times M_{\mathcal{C}_2}$ and

$$q = (q_{M_{\mathcal{C}_1}}, q_{M_{\mathcal{C}_2}}) \quad (8.30)$$

To prove (8.30), consider $(q_{E'}, q_{M_{\mathcal{C}_1}}, q_{M_{\mathcal{C}_2}})$, a reachable state of $E' \times M_{\mathcal{C}_1} \times M_{\mathcal{C}_2}$ and let $\alpha \in \Sigma$ be such that

$$\alpha \in \Sigma \implies (q_{E'}, q_{M_{\mathcal{C}_1}}, q_{M_{\mathcal{C}_2}}) \xrightarrow{\alpha} E' \times M_{\mathcal{C}_1} \times M_{\mathcal{C}_2} \quad .$$

Using the same argument as before, we deduce that $q_1 = \text{def } q_{M_{\mathcal{C}_1}}$ is not equal to $\top$ and thus belongs to $Q_1$, and that the pair $q = (q_1, q_2)$ is legal. Hence, we deduce, for $i = 1, 2$:

$$q_i \xrightarrow{\alpha} 1 \quad (8.29)$$

showing that $(q_1, q_2) \xrightarrow{\alpha} \mathcal{C}$ holds. Performing this move leads again to a pair of states belonging to $\leq (8.26)$, showing that $\leq (8.26)$ is a simulation relation.
Since, on the other hand, \( \preceq \) contains the pair consisting of the initial states, (8.30) follows. Hence, \( C \) simulates \( E \times M_1 \times M_2 \) in the sense of i/o-automata. (8.26) is thus proved, hence so is (8.24).

It remains to prove (8.25), that is:

\[ C \text{ is minimal in } C_{12} \text{ for refinement order.} \] (8.31)

If not, there would exist \( C' \in C_{12}, C' \preceq C, C' \neq C \), meaning that

\[ C' \text{ satisfies (4.2)} \] (8.32)

\[ E_{C'} \supseteq E_C \text{ and } M_{C'} \subseteq M_C \] (8.33)

and either \( E_{C'} \supset E_C \) or \( M_{C'} \subset M_C \) (8.34)

**Case** \( E_{C'} \supset E_C \) in (8.34): In this case \( E_{C'} \) simulates \( E_C \) but the converse is not true. By Lemma 8.4, there exists a word \( w \in \Sigma^* \), leading environments \( E_{C'} \) and \( E_C \) to states \( q' \) and \( q \), such that

\[ \Sigma_{E_{C'}}(q') \supset \Sigma_{E_C}(q) \] (8.35)

holds. Observe that (8.35) requires that \( q \neq \top \), the top state of components. Due to Condition 2 of Definition 8.3 and step (a) of the construction of the maximal environment in the same definition, (8.35) requires the existence of some action

\[ \alpha \in \Sigma_{E_{C'}}(q'), \alpha \notin \Sigma_{E_C}(q), \alpha \in \Sigma^{\text{out}} \]

which is a contradiction since, being a component, the environment \( E_C \) is receptive, meaning that \( \Sigma_{E_C}(q) \supseteq \Sigma^{\text{out}} \) for any \( q \). Thus we must have \( E_{C'} = E_C \). As a consequence,

\[ E_{C'} = E_C \] (8.36)

**Case** \( E_{C'} = E_C \) and \( M_{C'} \subset M_C \) in (8.34): By the characterization (4.2) of the contract composition in the meta-theory, \( M_{C'} \) must contain all compositions \( M_1 \times M_2 \), for \( M_1 \) and \( M_2 \) ranging over the sets of all implementations of \( C_1 \) and \( C_2 \). Now, checking the implementation relation \( M \models M' \) requires inspecting what happens at reachable states \( (q, q_M) \) of the composition \( E_C \times M \). In the pruning operation of step 4 of Definition 8.5, tentative transitions to exception states of the pre-composition \( C_1 \otimes C_2 \) are removed, which ensures
that, in the composition $E \times M$, only pairs $(q, q_M)$ in which $q$ is not an exception state of the pre-composition $C_1 \otimes C_2$, are accessed. On the other hand, at a reachable state $(\top, q_M)$ of the composition $E \times M$, ready sets of $M$ are unconstrained. To summarize:

- $M_{\mathcal{E}}$, must contain all compositions $M_1 \times M_2$, for $M_1$ and $M_2$ ranging over the sets of all implementations of $C_1$ and $C_2$;

- $M_{\mathcal{E}}$, must contain every component $M$ such that, at every state $(q, q_M)$ that is reachable in $E \times M$ and satisfies $q \neq \top$, we have: $\Sigma_M(q_M) \subseteq \Sigma_{\mathcal{E}}(q)$.

But these two properties exactly characterize the set $M_{\mathcal{E}}$ of all implementations of $\mathcal{E}$. Thus, $M_{\mathcal{E}} \supseteq M_{\mathcal{E}}'$ holds, a contradiction. This finishes the proof of (8.31), and the theorem is proved. \hfill \Box

In [64, 66] Bujtor and Vogler proved that the parallel composition of Interface Automata fails to be associative. The previous theorem shows that it is at least sub-associative. A more extensive discussion of associativity is provided in the next section about Modal Interfaces.

**Quotient:** In [53], incremental design of deterministic Interface Automata is studied.

Let $\mathcal{E}^1$ be the interface $\mathcal{E}$ with input and output actions interchanged. \hfill (8.37)

Given two Interface Automata $\mathcal{E}_1$ and $\mathcal{E}_2$, the greatest interface compatible with $\mathcal{E}_2$ such that their composition refines $\mathcal{E}_1$ is given by $(\mathcal{E}_1 \parallel \mathcal{E}_2^1)^1$.

**Dealing with variable alphabets:** So far we have presented the framework of interface automata for the case of a fixed alphabet. The clever reader may expect that dealing with variable alphabets can be achieved by using the mechanism of alphabet equalization via inverse projections.\footnote{The inverse projection of an i/o-automaton is simply achieved by adding, in each state, a self-loop for each missing symbol.} This is a correct guess for contract composition. It is however not clear if it is also adapted for conjunction for which no satisfactory construction exists as previously indicated.
In contrast, alphabet equalization and conjunction are elegantly addressed by the alternative framework of Modal Interfaces we develop now.

### 8.3 Modal Interfaces with fixed alphabet

Modal Interfaces inherit from both the Interface Automata and the originally unrelated notion of Modal Automaton (or Modal Transition System), see the bibliographical note in Section 8.8. As for Interface Automata, the semantics of Modal Interfaces is given below in terms of contracts, that is, pairs formed of a set of implementations and a set of valid environments, all of them represented as receptive i/o-automata. The presentation of Modal Interfaces we develop here is thus aligned with our meta-theory and, thus, differs from classical presentations. Again, we begin with the case of a fixed alphabet \( \Sigma \).

**Definition 8.6 (Modal Interface [227, 62]).** Call Modal Interface \( \mathcal{C} \) a tuple \( \mathcal{C} = (\Sigma^{\text{in}}, \Sigma^{\text{out}}, Q, \top, q_0, \rightarrow, \leftarrow) \),

where

1. \( \Sigma^{\text{in}}, \Sigma^{\text{out}}, Q, q_0 \) are as in i/o-automata,
2. \( q_0 \) may not belong to \( Q \),
3. \( \rightarrow, \leftarrow \subseteq Q \times \Sigma \times Q \) are two deterministic transition relations called *must* and *may*, respectively, satisfying the following consistency condition:
   \[
   q \xrightarrow{\alpha} q' \text{ implies } q \xrightarrow{\alpha} q'.
   \]  
   \( (8.38) \)
4. \( \top \in Q \) is a distinguished state called the *universal state* such that, for every action \( \alpha \in \Sigma \), \( \neg (\top \xrightarrow{\alpha}) \) and \( \top \xrightarrow{\alpha} \top \) both hold; besides self-loops, \( \top \) can only be the target of a may-transition labeled by an input action.

A Modal Interface \( \mathcal{C} \) such that \( q_0 \in Q \) and \( q_0 \neq \top \) induces two (possibly non receptive and non-reachable) i/o-automata:

\[
\mathcal{C}_{\text{must}} = (\Sigma^{\text{in}}, \Sigma^{\text{out}}, Q \setminus \{ \top \}, q_0, \rightarrow)
\]
and \( \mathcal{C}_{\text{may}} = (\Sigma^{\text{in}}, \Sigma^{\text{out}}, Q \setminus \{ \top \}, q_0, \leftarrow) \).

where \( \rightarrow \) and \( \leftarrow \) are implicitly restricted to \( Q \setminus \{ \top \} \).

\( \square \)
Besides the non-classical condition on the initial state, this definition is identical to that of [227], with a partial import from [62] regarding the universal state $\top$. Compared to [62], we make a minimal use of the distinguished universal state $\top$, related to compatibility. We insist that i/o-automata possess no distinguished state, so $\top$ cannot be a state of an i/o-automaton, whence the removal of it when defining $C^{\text{must}}$ and $C^{\text{may}}$.

In the sequel, whenever convenient, we mark the items of the tuple constituting $C$ by the subscript $C$, e.g., in $\top_C$ or $\rightarrow_C$. For pairs $(C_1, C_2)$ we only use the index 1, 2 for this subscript.

**Definition 8.7 (associated contract).** $C$ defines a contract according the meta-theory by fixing a pair $(E_C, M_C)$ as follows:

If $q_0 \notin Q$, we set by convention $M_C = \emptyset$ and $E_C = M$, thus making $C$ inconsistent.

If $q_0 = \top$, we set by convention $E_C = \emptyset$ and $M_C = M$, thus making $C$ incompatible.

Otherwise the following construction holds.

The set $E_C$ of the *legal environments* for $C$ collects all components $E$ satisfying the following conditions:

1. $\Sigma^\text{in}_E = \Sigma^\text{out}$ and $\Sigma^\text{out}_E = \Sigma^\text{in}$; consequently, $E$ and $C^{\text{must}}$, when seen as i/o-automata, are composable and the same holds with $C^{\text{may}}$ in lieu of $C^{\text{must}}$.

2. For any $\alpha \in \Sigma^\text{in}$ and state $q_E$ of $E$ such that $q_E \xrightarrow{\alpha} E$, and any reachable state $(q_E, q)$ of $E \times C^{\text{may}}$, it holds that $q \xrightarrow{\alpha} C^{\text{may}}$.

3. No state of the form $(q_E, \top)$ is reachable from $(q_{E,0}, q_0)$ in $E \times C^{\text{may}}$.

Define the particular environment $E'_C \in E_C$ as follows:

(a) We first consider $E'_C = (\Sigma^\text{out}, \Sigma^\text{in}, Q', q_0, \rightarrow')$, where $Q' = Q \setminus \{\top\}$, the restriction of $\rightarrow'$ to $Q' \times \Sigma^\text{in} \times Q'$ coincides with $\rightarrow$, and the restriction of $\rightarrow'$ to $Q' \times \Sigma^\text{out} \times Q'$ coincides with $\rightarrow$.

(b) We then make $E'_C$ receptive following (8.9) (page 231).

The set $M_C$ of the *implementations* of $C$ collects all components $M$ satisfying the following conditions:
4. $E \times E_{\text{may}}$ simulates $E \times M$ in the sense of (8.5), meaning that only 
may transitions are allowed for $E \times M$;

5. $E \times M$ simulates $E \times E_{\text{must}}$ in the sense of (8.5), meaning that must 
transitions are mandatory in $E \times M$. □

Observe that by construction the particular environment $E \in \mathcal{E}$ defined 
below (if it exists) is such that $E \times E_{\text{may}}$ simulates $E \times E_{\text{may}}$ in the sense of 
(8.5) (page 230) for any $E \in \mathcal{E}$.

**Summary of Results 3.**

1. Through Definition 8.7, the framework of Modal Interfaces equipped 
with its existing operations of modal refinement, conjunction as GLB, 
parallel composition, and quotient, instantiates the meta-theory.

2. The parallel composition is not associative but only sub-associative.

3. Variable alphabets are conveniently supported.

4. Assume/Guarantee reasoning can be emulated.

5. Synthesis methods exist to derive subcontracts from a system-level 
contract and the SysML-like specification of a system architecture. □

The rest of this section develops the lemmas and theorems supporting 
these results.

**8.3.1 Miscellaneous**

Observe first that, since components are receptive i/o-automata, we can equival-
ently replace $\alpha \in \Sigma_M$ by $\alpha \in \Sigma^\text{out}_M$ in the above condition 5. On the other hand, 
the consideration of the particular environment $E \in \mathcal{E}$ is justified by the follow-
ing result:

**Lemma 8.7.** The environment $E \in \mathcal{E}$ is maximal in $\mathcal{E}$ with respect to simula-
tion relation (8.5). □

**Proof.** By construction, $E' \times E_{\text{may}}$ is isomorphic to $(\emptyset, \Sigma, Q', q_0, \rightarrow)$, i.e., it 
is obtained from $E'$ by simply turning inputs to outputs. Consequently, Condition 2 holds for $E'$. Due to Condition 4 of Definition 8.6, Condition 3 holds
for $E'_E$. To show the maximality of $E'_E$, we consider an arbitrary environment $E$ of $\mathcal{E}$ and we define the following relation $\leq^e \subseteq Q_E \times (Q' \cup \{\top\})$:

$$\leq^e = \{(q_E, q) | (q_E, q) \text{ is reachable in } E \times \mathcal{E}^{\text{may}}\} \cup (Q_E \times \{\top\})$$

We claim that $\leq^e$ is a simulation relation from $E'_E$ to $E$. (8.39)

First of all the top state $\top$ trivially simulates any state from $Q_E$. It is thus enough to consider pairs $q_E \leq^e q$ such that $q \neq \top$, meaning that $(q_E, q)$ is reachable in $E \times \mathcal{E}^{\text{may}}$. Next, let $\alpha \in \Sigma$ be such that $q_E \xrightarrow{\alpha} \top$ holds. If $\alpha \in \Sigma^{\text{in}}$ then we know by Condition 2 of Definition 8.3 that $q \xrightarrow{\alpha} \mathcal{E}^{\text{must}}$ holds, which implies $q \xrightarrow{\alpha} E'_E$ by construction of $E'_E$. If $\alpha \in \Sigma^{\text{out}}$ then $q \xrightarrow{\alpha} E'_E$ also holds since $E'_E$ is receptive (recall that $\Sigma^{\text{out}}$ is the set of input actions of the environment $E'_E$). Performing this move leads $E'_E$ either to a state belonging to $Q'$, or to $\top$. Thus, performing an $\alpha$-move in both $E$ and $E'_E$ leads to a unique pair $(q'_E, q')$ that either is reachable in $E \times \mathcal{E}^{\text{may}}$ or belongs to $Q_E \times \{\top\}$. In any case the new pair $(q'_E, q')$ again belongs to $\leq^e$. This proves (8.39) and the lemma.

In addition to the maximal environment $E'_E$ associated to contract $\mathcal{E}$, we will need its maximal implementation $M'_E$ and minimal implementation $m'_E$, constructed as follows, respectively:

- **Construction of $M'_E$:**
  
  (a) We first consider $M'_E$, which is obtained by $\mathcal{E}^{\text{may}}$, seen as an i/o-automaton;
  
  (b) We then obtain $M'_E$ by making $M'_E$ receptive following (8.9).

- **Construction of $m'_E$:**
  
  (a) We first consider $m'_E = \mathcal{E}^{\text{must}}$, seen as an i/o-automaton;
  
  (b) We then obtain $m'_E$ by making $m'_E$ minimally receptive as follows: for each state $q$ of $m'_E$ and each action $\alpha \in \Sigma^{\text{in}}$ such that $\alpha \notin \Sigma^{\text{in}}(q)$, add a transition $q \xrightarrow{\alpha} \top^w$. 

---

8.3. Modal Interfaces with fixed alphabet

245
where $\top^w$ is a weak top state satisfying:
\[
\forall \alpha \in \Sigma^\text{in} \implies \top^w \xrightarrow{\alpha} \top^w \quad \text{and} \quad \forall \alpha \in \Sigma^\text{out} \implies \top^w \xrightarrow{\alpha}
\] (8.40)

Conditions 4 and 5 of Definition 8.7 are satisfied by both $M_\emptyset$ and $m_\emptyset$. The following result justifies this construction:

**Lemma 8.8.** $M_\emptyset$ and $m_\emptyset$ are respectively maximal and minimal in $M_\emptyset$ with respect to simulation relation (8.5). Furthermore, every component $M$ that simulates $m_\emptyset$ and is simulated by $M_\emptyset$ is an implementation of $\emptyset$. \(\square\)

**Proof.** By construction, recalling that $Q'$ is the set of non-top states of $E_\emptyset$,
\[
E_\emptyset \times M_\emptyset \quad \text{is isomorphic to} \quad (\emptyset, \Sigma, Q', q_0, \rightarrow')
\]
and
\[
E_\emptyset \times m_\emptyset \quad \text{is isomorphic to} \quad (\emptyset, \Sigma, Q', q_0, \rightarrow)
\]
where $\rightarrow'$ is the transition relation of $E'_\emptyset$ and $\rightarrow$ is the must transition relation of $\emptyset$. Therefore, Conditions 4 and 5 of Definition 8.7 are satisfied by both $M_\emptyset$ and $m_\emptyset$.

We now prove the maximality of $M_\emptyset$. We consider an arbitrary implementation $M$ and we define the following relation $\leq^u \subseteq Q_M \times (Q' \cup \{T\})$:
\[
\leq^u = \{(q_M, q) | (q_M, q) \text{ is reachable in } M \times \emptyset^\text{may}\} \cup (Q_M \times \{T\})
\]
We claim that
\[
\leq^u \text{ is a simulation relation from } M_\emptyset \text{ to } M. \quad \text{(8.41)}
\]
Since the top state $T$ is universal for simulation, it is enough to consider pairs $q_M \leq^u q$ such that $q \neq T$, which implies that $(q_M, q)$ is reachable in $M \times \emptyset^\text{may}$. Let $\alpha \in \Sigma$ be such that
\[
q_M \xrightarrow{\alpha} q_M'
\]
By Condition 4 of Definition 8.7, we must have $q \xrightarrow{\alpha} q'$ for some $q' \in Q'$, which implies
\[
q \xrightarrow{\alpha} q'
\]
by construction of $M_\emptyset$ and $(q'_M, q')$ is again reachable in $M \times \emptyset^\text{may}$. Hence $q'_M \leq^u q'$ still holds. Since $q_{M,0} \leq^u q_0$ holds, (8.41) follows.

We next prove the minimality of $m_\emptyset$. We consider an arbitrary implementation $M$ and we define the following relation $\leq^m \subseteq (Q' \cup \{\top^w\}) \times Q_M$:
\[
\leq^m = \{(q, q_M) | (q, q_M) \text{ is reachable in } \emptyset^\text{must} \times M \} \cup (\{\top^w\} \times Q_M)
\]
We claim that

\[ \leq_m \] is a simulation relation from \( M \) to \( m_\mathcal{C} \). \hfill (8.42)

The state \( \top_w \) is minimal with respect to simulation for receptive i/o-automata. Therefore, it is enough to consider pairs \( q \leq_m q_M \) such that \( q \neq \top_w \), which implies that \((q, q_M)\) is reachable in \( \mathcal{C}^{\text{must}} \times M \). Let \( \alpha \in \Sigma \) be such that

\[ q \xrightarrow{\alpha} q' \]

Suppose \( q' = \top_w \). Then, by construction of the minimal implementation \( m_\mathcal{C} \), this is only possible if \( \alpha \in \Sigma^{\text{in}} \), in which case \( q_M \xrightarrow{\alpha} q'_M \) also holds for some \( q'_M \) since \( M \) is receptive, and \( \top_w \leq_m q'_M \) still holds. Consider next the case \( q' \neq \top_w \). Then by construction of the minimal implementation, we must have \( q \xrightarrow{\alpha} q' \), which, by Condition 5 or Definition 8.7, implies

\[ q_M \xrightarrow{\alpha} q'_M \]

for some \( q'_M \in Q_M \) and \( (q', q'_M) \) is reachable in \( \mathcal{C}^{\text{must}} \times M \). Therefore, \( q' \leq_m q'_M \) still holds, which proves (8.42).

\[ \square \]

**Introducing must and may sets:** It will be useful for the mathematics to reformulate the conditions of Definition 8.7 using *must* and *may* sets we introduce now. For \( \mathcal{C} \) a Modal Interface and \( q \) a state of it, we introduce the following *may* and *must* sets:

\[ \text{may}_\mathcal{C}(q) = \{ \alpha \in \Sigma \mid q \xrightarrow{\alpha} \} \] \hfill (8.43)

\[ \text{and } \text{must}_\mathcal{C}(q) = \{ \alpha \in \Sigma \mid q \xrightarrow{\alpha} \} \]

and the inclusion \( \text{must}_\mathcal{C}(q) \subseteq \text{may}_\mathcal{C}(q) \) holds, by (8.38). For both notions, we omit the subscript \( M \) or \( \mathcal{C} \) when no confusion can result. The following lemma holds:

**Lemma 8.9.** The conditions of Definition 8.7 can be reformulated as follows:

**Condition 2:** For every pair \((q_E, q)\) that is reachable in \( E \times \mathcal{C}^{\text{may}} \), we have:

\[ \Sigma^{\text{in}} \cap \Sigma_E(q_E) \subseteq \text{must}(q) \] \hfill (8.44)

**Condition 3:** Unchanged.
**Condition 4:** For every pair \((q, q_M)\) that is reachable in \(E_\mathcal{E} \times M\), we have:

\[
\Sigma_{\text{out}} \cap \Sigma_M(q_M) \subseteq \Sigma_{\text{out}} \cap \text{may}(q) \quad (8.45)
\]

**Condition 5** For every pair \((q, q_M)\) that is reachable in \(E_\mathcal{E} \times M\), we have:

\[
\Sigma_{\text{out}} \cap \text{must}(q) \subseteq \Sigma_{\text{out}} \cap \Sigma_M(q_M) \quad (8.46)
\]

**Proof.** The reformulation of Condition 2 is immediate. Condition 4 is equivalent to the following inclusion, which by itself implies that the top state \(\top\) of \(E_\mathcal{E}\) is not reachable in the product \(E_\mathcal{E} \times M\):

\[
\left(\Sigma_{\text{in}} \cap \text{must}(q)\right) \cup \Sigma_{\text{out}} \cap \Sigma_M(q_M) \subseteq \left(\Sigma_{\text{in}} \cap \text{must}(q)\right) \cup \left(\Sigma_{\text{out}} \cap \text{may}(q)\right) \quad (8.47)
\]

Using the partitioning \(\Sigma = \Sigma_{\text{in}} \cup \Sigma_{\text{out}}\), inclusion (8.47) is equivalent to the conjunction of the following two inclusions:

\[
\begin{align*}
\Sigma_{\text{in}} \cap \text{must}(q) \cap \Sigma_M(q_M) & \subseteq \Sigma_{\text{in}} \cap \text{must}(q) \\
\Sigma_{\text{out}} \cap \Sigma_M(q_M) & \subseteq \Sigma_{\text{out}} \cap \text{may}(q)
\end{align*}
\]

which is equivalent to (8.45) since the first inclusion is a tautology. The reasoning for Condition 5 is similar. First, Condition 5 is equivalent to

\[
\text{must}(q) \subseteq \left(\Sigma_{\text{out}} \cap \text{may}(q)\right) \cup \Sigma_{\text{in}} \cap \Sigma_M(q_M)
\]

which, by decomposing \(\Sigma = \Sigma_{\text{in}} \cup \Sigma_{\text{out}}\) and using (8.45), proves (8.46). \(\square\)

### 8.3.2 Relation of Definitions 8.6 and 8.7 with the existing semantics of Modal Interfaces

The first sentence of Definition 8.6 is a verbatim of the original definition of Modal Interfaces [227]. As for Interface Automata in Section 8.2, the handling of the initial state \(q_0\) is heterodox and motivated by our aim that Definition 8.7 casts Modal Interfaces in the framework of the meta-theory. For the same reason, Definition 8.7 introduces the two sets \(E_\mathcal{E}\) and \(M_\mathcal{E}\), whereas the classical theory of Modal Interfaces considers and develops a different notion of model (often also called “implementation”, which is unfortunate). Nevertheless, the following relation holds between \(M_\mathcal{E}\) and the set of models of \(\mathcal{E}\).
Theorem 8.10. Assume \( q_0 \in Q \). Then:

1. The map \( M \to \overline{M} \) defined in (8.9) maps every model of \( \mathcal{C} \) to an implementation of \( \mathcal{C} \), i.e., \( \overline{M} \in \mathcal{M}_{\mathcal{E}} \).

2. Vice-versa, for every implementation \( M \in \mathcal{M}_{\mathcal{E}} \),

\[
N = \left( \Sigma^\text{in}, \Sigma^\text{out}, Q_M \times Q', (q_M, 0), \overline{M} \times E_{\mathcal{C}} \right)
\]

is a model of \( \mathcal{C} \), where we recall that \( Q' = Q \setminus \top \) is the set of states of the maximal environment \( E_{\mathcal{C}} \).

Proof. Statement 1: We prove that every model \( M = (\Sigma^\text{in}, \Sigma^\text{out}, Q_M, q_0^M, \overline{M}) \) of \( \mathcal{C} \) yields an implementation \( \overline{M} \) of \( \mathcal{C} \) according to Definition 8.7. Recall that \( M \models \mathcal{C} \) — written \( M \models \mathcal{C} \) — holds if and only if, for every pair of states \((q_M, q)\) that is reachable in \( M \times \mathcal{C}^\text{may} \), the following holds [227]:

\[
\text{must}(q) \subseteq \Sigma_M(q_M) \subseteq \text{may}(q) .
\]  

(8.48)

Using Lemma 8.9, this condition implies that Conditions 4 and 5 of Definition 8.7 hold for \( M \). Replacing \( M \) by its receptive counterpart \( \overline{M} \) does not change anything since the transitions added when moving from \( M \) to \( \overline{M} \) are canceled in the composition \( E_{\mathcal{C}} \times M \).

Statement 2: We prove that, for every triple \((q_M, q, q)\) of states that is reachable in \( N \times \mathcal{C}^\text{may} \),

\[
\text{must}(q) \subseteq \Sigma_N(q_M, q) \subseteq \text{may}(q)
\]  

(8.49)

By construction, we have

\[
\Sigma_N(q_M, q) = \Sigma_M(q_M) \cap \left( \left( \Sigma^\text{in} \cap \text{must}(q) \right) \cup \left( \Sigma^\text{out} \cap \text{may}(q) \right) \right)
\]  

(8.50)

Using again Lemma 8.9, Conditions 4 and 5 of Definition 8.7 for \( M \), imply: for every pair \((q, q_M)\) of states that is reachable in \( E_{\mathcal{C}} \times M \),

\[
\Sigma^\text{out} \cap \text{must}(q) \subseteq \Sigma^\text{out} \cap \Sigma_M(q_M) \subseteq \Sigma^\text{out} \cap \text{may}(q)
\]  

(8.51)

Intersecting the second inclusion of (8.51) with \( \text{may}(q) \) and using (8.50) yields

\[
\Sigma^\text{out} \cap \Sigma_N(q_M, q) \subseteq \Sigma^\text{out} \cap \text{may}(q)
\]  

(8.52)
On the other hand, intersecting the first inclusion of (8.51) with \( \text{must}(q) \) and using (8.50) and the fact that \( \text{must}(q) \subseteq \text{may}(q) \) gives

\[
\Sigma^{\text{out}} \cap \text{must}(q) \subseteq \Sigma^{\text{out}} \cap \Sigma_N(q_M, q) \tag{8.53}
\]

Finally, using once more (8.50) and the fact that \( M \) is receptive, we get
\[
\Sigma^{\text{in}} \cap \Sigma_N(q_M, q) = \Sigma^{\text{in}} \cap \text{must}(q),
\]
which, together with (8.52) and (8.53), yields (8.49).

8.3.3 Pseudo-Modal Interfaces and their reduction

The introduction of the operations on modal interfaces will require weakening the consistency condition (8.38) to

\[
q^\alpha \rightarrow q' \quad \text{and} \quad q^\alpha \dashv q'' \quad \text{implies} \quad q' = q''. \tag{8.54}
\]

A tuple \( \mathcal{C} \) according to Definition 8.7 where the consistency condition (8.38) is weakened to (8.54) is called a **Pseudo-Modal Interface**. We recall here the operation of

\[
\text{reduction} : \mathcal{C} \rightarrow [\mathcal{C}], \tag{8.55}
\]

which maps any Pseudo-Modal Interface \( \mathcal{C} \) to a unique Modal Interface \([\mathcal{C}]\) possessing the same set of models, by minimally pruning the former [227]. Say that state \( q \in Q \) of \( \mathcal{C} \) is **consistent** if \( q^\alpha \rightarrow q' \) implies \( q^\alpha \dashv q'' \), otherwise we say that it is **inconsistent**. Assume that \( \mathcal{C} \) has some inconsistent state \( q \in Q \), meaning that, for some action \( \alpha \), \( q^\alpha \rightarrow q' \) holds but \( q^\alpha \dashv q'' \) does not hold. Inconsistent states can be pruned away from \( Q \) without changing the set of models of \( \mathcal{C} \). More precisely, all may transitions leading to \( q \) can be deleted from \( \mathcal{C} \) without changing its set of models. Performing this makes state \( q \) unreachable in \( \mathcal{C}^{\text{may}} \). Since we have removed may transitions, some more states have possibly become inconsistent. So, we must repeat the same procedure: set \( Q_0 = Q \) and for \( n \geq 0 \), denote by \( Q_{n+1} \) the set obtained by removing all inconsistent states of \( Q_n \). Since the number of states is finite, this procedure eventually reaches a fixpoint that we call \( Q_{\text{con}} \) and the set \( Q \) of states partition as \( Q = Q_{\text{con}} \cup Q_{\text{incon}} \), where \( Q_{\text{incon}} \) collects all states that were or became inconsistent as a result of this procedure. \( Q_{\text{con}} \) only collects consistent states. In addition, since the fixpoint has been reached, \( Q_{\text{incon}} \) is unreachable from \( Q_{\text{con}} \). As a final step, we denote by \([\mathcal{C}]\) the Modal Interface
obtained by restricting $\mathcal{C}$ to its subset of states $Q_{\text{con}}$ and adjusting the may and must transition relations accordingly. If $\mathcal{C}$ yields a consistent contract if and only if $q_0 \in Q_{\text{con}}$. In the sequel, unless otherwise specified, we will only consider reduced Modal Interfaces.

### 8.3.4 Refinement and conjunction

Conjunction and refinement are instantiated in a very elegant way in the theory of Modal Interfaces. Contract refinement in the sense of the meta-theory is instantiated by the effective notion of Modal refinement we introduce now. Roughly speaking, modal refinement consists in enlarging the must relation (thus enlarging the set of legal environments) and restricting the may relation (thus restricting the set of implementations). The formalization requires the use of simulation relations.

**Definition 8.8** (modal refinement [227, 62]). Let $\mathcal{C}_i, i = 1, 2$ be two Modal Interfaces and $q_i$ be a state of $\mathcal{C}_i$, for $i = 1, 2$. Say that $q_2$ modal refines $q_1$, written $q_2 \preceq q_1$, if $q_2 \notin Q_2$, else if $q_1 = \top$, else if the following holds:

$$
\begin{align*}
\text{may}_2(q_2) &\subseteq \text{may}_1(q_1), \quad \text{must}_2(q_2) \supseteq \text{must}_1(q_1) \\
\forall \alpha \in \Sigma : \quad q_1 \stackrel{\alpha}{\rightarrow} q_1' \\
q_2 \stackrel{\alpha}{\rightarrow} q_2' &\implies q_2' \preceq q_1'
\end{align*}
$$

(8.56)

Say that $\mathcal{C}_2 \preceq \mathcal{C}_1$ if $q_{2,0} \preceq q_{1,0}$.

Observe that, by decree, any state refines the universal state [62]. The following result relates modal refinement with contract refinement as defined in the meta-theory. It justifies the consideration of modal refinement:

**Theorem 8.11.** Let $\mathcal{C}_i, i = 1, 2$ be two Modal Interfaces such that $\Sigma_{i}^{\text{in}} = \Sigma_{2}^{\text{in}}$, implying $\Sigma_{i}^{\text{out}} = \Sigma_{2}^{\text{out}}$ (we denote them by $\Sigma^{\text{in}}$ and $\Sigma^{\text{out}}$, respectively). Then:

$$
q_{2,0} \preceq q_{1,0} \quad \text{if and only if} \quad \begin{cases} 
\mathcal{E}_{\mathcal{C}_2} \supseteq \mathcal{E}_{\mathcal{C}_1} \\
\mathcal{M}_{\mathcal{C}_2} \subseteq \mathcal{M}_{\mathcal{C}_1}
\end{cases}
$$

Proof. As a preliminary reasoning step, consider first the case when $q_{2,0} \notin Q_2$, meaning that contract $\mathcal{C}_2$ is inconsistent, $\mathcal{M}_{\mathcal{C}_1} = \emptyset$, and possesses a trivial set of environments, $\mathcal{E}_{\mathcal{C}_1} = \mathcal{M}$. Clearly the right hand side holds whatever $\mathcal{C}_1$
is. Second, we consider the case in which \( q_{1,0} = \top \), meaning that \( C_1 \) is incompatible \( \mathcal{E}_{\mathcal{E}_1} = \emptyset \) and possesses a trivial set of implementations \( M_{\mathcal{E}_1} = \mathcal{M} \). Clearly the right hand side holds whatever \( C_2 \) is. Thus, for the rest of the proof we can assume that \( q_{1,0} \neq \top \) and \( q_{2,0} \in Q_2 \).

We begin with the “only if” part. Observe that if \( C_1 \) is incompatible, then \( \mathcal{E}_{\mathcal{E}_1} = \emptyset \) and \( M_{\mathcal{E}_1} = \mathcal{M} \) by Definition 8.7 and thus the right hand side of the iff statement holds whatever \( C_2 \) is. For the rest of the proof of the “only if” part, we thus assume \( \mathcal{E}_{\mathcal{E}_1} \neq \emptyset \), hence the maximal environment \( \mathcal{E}_{\mathcal{E}_1} \) exists. As a prerequisite, note the following:

- if \( E \models^E C \) and \( E \) simulates \( E' \) in the sense of i/o-automata, then \( E' \models^E C \) follows;
- if \( M \) satisfies Condition 4 of Definition 8.7 and \( M \) simulates \( M' \) in the sense of i/o-automata, then \( M' \) too satisfies Condition 4 of Definition 8.7;
- if \( M \) satisfies Condition 5 of Definition 8.7 and \( M \) simulates \( M' \) in the sense of i/o-automata, then \( M' \) too satisfies Condition 5 of Definition 8.7.

Consequently, the only if part amounts to proving

\[
q_{2,0} \preceq q_{1,0} \implies \begin{cases} E_{\mathcal{E}_1} \models^E C_2 \\ M_{\mathcal{E}_2} \text{ satisfies Condition 4 of Definition 8.7 relative to } C_1 \\ m_{\mathcal{E}_2} \text{ satisfies Condition 5 of Definition 8.7 relative to } C_1 \end{cases}
\]  

(8.57)

Focus on the environments. Pick a pair \((q_{E,\mathcal{E}_1}, q_2)\) that is reached in \( E_{\mathcal{E}_1} \times \mathcal{E}_{\mathcal{E}_2} \) by performing some word \( w \in \Sigma^* \) and let \( \alpha \in \Sigma^\text{in} \) be such that

\[
q_{E,\mathcal{E}_1} \xrightarrow{\alpha} q'_{E,\mathcal{E}_1} \quad \quad \text{(8.58)}
\]

Suppose that \( q_{E,\mathcal{E}_1} = \top \). Since state \( \top \) is absorbing, we can consider \( v \), the largest prefix of \( w \) leading both \( E_{\mathcal{E}_1} \) and \( \mathcal{E}_{\mathcal{E}_1} \) to a state \( q'_1 \neq \top \) and \( \mathcal{E}_{\mathcal{E}_2} \) to a state \( q''_2 \), and let \( \alpha' \in \Sigma^\text{out} \) (it has to be an output) be such that

\[
q'_1 \xrightarrow{\alpha'} q''_1 \xrightarrow{\text{out}} \top
\]
Then, by construction of $E_{\mathcal{C}_1}$ we have $q_1 '' \xrightarrow{\alpha} q_1$, which, since $\text{may}_2(q_2 '') \subseteq \text{may}_1(q_1 '')$, implies

$$q_2 '' \xrightarrow{\alpha} q_2.$$ 

But this contradicts the assumption that pair $(q_{E_{\mathcal{C}_1}}, q_2)$ is reached in $E_{\mathcal{C}_1} \times \mathcal{C}_2^{\text{may}}$ by performing $w \in \Sigma^*$. Consequently, $q_{E_{\mathcal{C}_1}} \neq \top$. Then, by construction of $E_{\mathcal{C}_1}$, (8.58) implies $q_1,0 \xrightarrow{w} q_1$ with $q_1 = q_{E_{\mathcal{C}_1}}$, hence $q_2 \leq q_1$ follows. By Condition 2 of Definition 8.7 relative to $\mathcal{C}_1$:

$$q_{E_{\mathcal{C}_1}} \xrightarrow{\alpha} q_{E_{\mathcal{C}_1}}', q_{E_{\mathcal{C}_1}}' \text{ implies } q_1 \xrightarrow{\alpha} q_1'$$

(8.59)

which, since $q_2 \leq q_1$, implies $q_2 \xrightarrow{\alpha} q_2'$. We thus proved that (8.58) implies $q_2 \xrightarrow{\alpha} q_2'$, which, since $\Sigma_2^{in} = \Sigma_2^{in}$, proves that $E_{\mathcal{C}_1} \models \mathcal{C}_2$ in the right hand side of (8.57). The properties concerning the implementations in the right hand side of (8.57) are proved similarly: the property of the maximal implementation is proved by working with $\mathcal{C}_2^{\text{may}}$ and the property of the minimal implementation is proved by working with $\mathcal{C}_2^{\text{must}}$.

We now move to the “if” part of the theorem. Again we first consider the case in which $E_{\mathcal{C}_1} = \emptyset$, which implies that $\mathcal{C}_1$ is incompatible and has trivial set of implementations: $M_{\mathcal{C}_1} = \mathcal{M}$. This is equivalent to $q_{1,0} = \top$ and we know that the universal state is refined by any state. The “if” statement holds in this case. We can now focus on the case $E_{\mathcal{C}_1} \neq \emptyset$. From the right hand side of the statement of the theorem, we infer that

$$E_{\mathcal{C}_2} \text{ simulates } E_{\mathcal{C}_1},$$

$$M_{\mathcal{C}_1} \text{ simulates } M_{\mathcal{C}_2} \text{ and }$$

$$m_{\mathcal{C}_2} \text{ simulates } m_{\mathcal{C}_1}$$

(8.60)

and we denote by $\leq^e$, $\leq^u$, and $\leq^m$ the corresponding simulation relations. We claim that the following relation $\leq$ is an alternating simulation, from $\mathcal{C}_2$ by $\mathcal{C}_1$. Consider the following relation $\leq \subseteq Q_2 \times Q_1$: for $(q_2, q_1) \in Q_2 \times Q_1$,

$$q_2 \leq q_1 \iff \begin{cases} 
q_1 \leq^e q_2 \text{ and } \\
q_2 \leq^u q_1 \text{ and } \\
q_1 \leq^m q_2
\end{cases}$$

(8.61)

A mild extension of the proof that (8.16) was an alternating simulation shows that the relation defined in (8.61) is a modal refinement and $q_{2,0} \leq q_{1,0}$ holds. This proves the theorem. □
The conjunction of two Modal Interfaces is thus the Greatest Lower Bound (GLB) with respect to modal refinement order. Its computation proceeds in two steps. In a first step, we wildly consider the pre-conjunction by taking union of must sets and intersection of may sets, and then we reduce the so obtained pseudo-Modal Interface:

**Definition 8.9.** The pre-conjunction\(^4\) \(\mathcal{C}_1 \land \mathcal{C}_2\) of two Modal Interfaces is defined if and only if \(\Sigma^\text{in}_1 = \Sigma^\text{in}_2\) and \(\Sigma^\text{out}_1 = \Sigma^\text{out}_2\) and is the Pseudo-Modal Interface given by \(\Sigma^\text{in} = \Sigma^\text{in}_1\), \(\Sigma^\text{out} = \Sigma^\text{out}_1\), \(\mathcal{Q} = (Q_1 \times Q_2) \cup \{\bot\}\), \(q_0 = (q_{1,0}, q_{2,0})\), \(\top \bot = (\top \bot 1, \top \bot 2)\), and the two transition relations:

\[
(q_1, q_2) \xrightarrow{\alpha} (q'_1, q'_2) \text{ iff } q_i \xrightarrow{\alpha_i} q'_i, \text{ for } i = 1, 2
\]

\[
(q_1, q_2) \xrightarrow{\bot} \text{ iff } q_i \xrightarrow{\bot_i} q'_i \text{ and } \alpha \notin \text{must}_j(q_j), \text{ for } i, j = 1, 2, j \neq i
\]

\[
(q_1, q_2) \xrightarrow{\bot} (q'_1, q'_2) \text{ iff } q_i \xrightarrow{\bot_i} q'_i, \text{ for } i = 1, 2
\]

By construction, the must and may sets of \(\mathcal{C}_1 \land \mathcal{C}_2\) are given by:

\[
\text{must}(q_1, q_2) = \text{must}_1(q_1) \cup \text{must}_2(q_2)
\]

\[
\text{may}(q_1, q_2) = \text{may}_1(q_1) \cap \text{may}_2(q_2)
\]

(8.62)

Now by (8.62), we can see that \(\mathcal{C}_1 \land \mathcal{C}_2\) may involve inconsistent states and, thus, in a second step, the pruning introduced in (8.55) must be applied:

\[
\mathcal{C}_1 \land \mathcal{C}_2 = [\mathcal{C}_1 \land \mathcal{C}_2]
\]

(8.63)

Say that the two Modal Interfaces \(\mathcal{C}_1\) and \(\mathcal{C}_2\) are consistent if \(\mathcal{C}_1 \land \mathcal{C}_2\) is consistent in the sense of Definition 8.7.

### 8.3.5 Parallel composition

For Modal Interfaces, we are able to define both parallel composition and quotient in the sense of the meta-theory. As explained in [62], the parallel composition of Modal Interfaces originally proposed in [227] is not associative (despite it was erroneously claimed so in this reference). Still, the framework of [227] is interesting as we shall see. For correctness, we thus need to introduce the \(n\)-ary parallel composition. As it was the case for Interface Automata, parallel composition for Modal Interfaces raises compatibility issues, thus, a two-step procedure is again followed for its construction.

\(^4\)Pre-conjunction was originally denoted by the symbol \& in [225, 228, 227].
Definition 8.10. The pre-composition $\bigotimes_{i \in I} C_i$ of a tuple of Modal Interfaces is only defined if the output alphabets $\Sigma_{i}^{\text{out}}, i \in I$, are pairwise disjoint. It is the Modal Interface given by: $\Sigma^{\text{out}} = \bigcup_{i \in I} \Sigma_{i}^{\text{out}}, Q = (\prod_{i \in I} Q_{i}) \cup \{\top\}$ where the fresh state $\tau_{I}$ is the universal state of the pre-composition, $q_0 = (q_{i,0}, i \in I)$, and the two transition relations:

$$(q_{i}, i \in I) \xrightarrow{\alpha} (q'_{i}, i \in I) \text{ iff } \forall i \in I: q_{i} \xrightarrow{\alpha} q'_{i}$$

$$(q_{i}, i \in I) \xrightarrow{\alpha} (q'_{i}, i \in I) \text{ iff } \forall i \in I: q_{i} \xrightarrow{\alpha} q'_{i}$$

By construction, the must and may sets of $\bigotimes_{i \in I} C_i$ are given by:

$$\text{must}(q_{i}, i \in I) = \bigcap_{i \in I} \text{must}_{i}(q_{i})$$

$$\text{may}(q_{i}, i \in I) = \bigcap_{i \in I} \text{may}_{i}(q_{i})$$

(8.64)

Say that a state $q = (q_{i}, i \in I)$ of $\bigotimes_{i \in I} C_i$ is illegal if

$$\exists i, j \in I, \exists \alpha \in \Sigma_{i}^{\text{out}} : \alpha \in \text{may}_{i}(q_{i}) \text{ and } \alpha \notin \text{must}_{j}(q_{j})$$

(8.65)

When the pre-composition is in an illegal state, some implementation of all but the $j$-th interfaces may output an action that is not accepted by an implementation of the $j$-th interface — this situation would contradict the objective of independent implementation. Hence we expect a valid environment to prevent the implementation from reaching an illegal state. This is ensured by applying the following post-processing.

Define the subset $E \subseteq Q$ of exception states as being the smallest set of states $q \in Q$ satisfying the following properties:

- if $q$ is illegal, then $q \in E$;

- if $q \xrightarrow{\alpha} q'$ holds for some $\alpha \in \Sigma^{\text{out}}$ and $q' \in E$, then $q \in E$.

(8.66)

Exception states need therefore to be pruned away from $\bigotimes_{i \in I} C_i$. To this end, we perform the following by using the universal state $\tau_{I}$:

If $q_0 = (q_{1,0}, q_{2,0})$ is illegal, then we redefine $q_0 \leftarrow \tau_{I}$, thus making the universal state the only reachable state.

(8.67)

Otherwise, we perform the following transformation on the pre-composition: for every exception state $q$ and input action $\alpha \in \Sigma^{\text{in}}$ such that there exists $q'$ reachable in $\bigotimes_{i \in I} C_i^{\text{may}}$ and satisfying $q' \xrightarrow{\alpha} q$, we replace $q' \xrightarrow{\alpha} q$ by $q' \xrightarrow{\alpha} \tau_{I}$.

(8.68)
Definition 8.11. Definition 8.10 completed with the post-processing (8.67) or (8.68), yields the contract composition \( \bigotimes_{i \in I} C_i \). If post-processing (8.67) applies, then the resulting composition is incompatible. Otherwise, the resulting composition no longer possesses illegal states. \( \square \)

The above construction is justified by the following result:

Theorem 8.12. \( \bigotimes_{i \in I} C_i \) as defined in Definition 8.11 instantiates the contract composition from the meta-theory. As a consequence, contract composition is sub-associative. \( \square \)

Proof. We use the characterization (4.1),(4.2) of the contract composition in the meta-theory and we give only the proof for two contracts, for the sake of readability. Accordingly, the universal state is denoted by \( \top_{12} \). Our objective is thus to prove that \( C_1 \otimes C_2 \) as in Definition 8.11 coincides with the contract characterized by conditions (4.1),(4.2).

As a preliminary step, we consider the case when (8.67) applies, i.e., when \((q_{1,0}, q_{2,0})\) is illegal. We know that incompatibility follows in the existing Modal Interface theory [227, 62]. Making \( \top_{12} \) the initial state makes \( C \) incompatible according to Definition 8.7, see the first sentence following the construction of \( E_C \) therein. Thus, the theorem holds in this case.

We now focus on the other case and first prove that \( C = \text{def} \ C_1 \otimes C_2 \) belongs to the set \( C_{12} \) introduced in (4.1). (8.69)

Consider an arbitrary triple \((E, M_1, M_2)\) where \( E \) is an environment of \( C \), \( M_1 \) is an implementation of \( C_1 \), and \( M_2 \) is an implementation of \( C_2 \). We need to prove the following:

\[
M_1 \times M_2 \models^n C \quad \text{(8.70)}
\]

\[
E \times M_2 \models^*_1 C_1, \quad \text{(8.71)}
\]

together with the symmetric property obtained by exchanging indices 1 and 2. In the following we repeatedly use the characterization formulated in Lemma 8.9. First, \( E \), being an environment of \( C \), satisfies:

\[
\text{for any reachable state } (q_E, q) \text{ of } E \times C_{\text{may}}, \text{ we have } \Sigma^{\text{in}} \cap \Sigma_{E(q_E)} \subseteq \text{must}_C(q). \quad \text{(8.72)}
\]
8.3. Modal Interfaces with fixed alphabet

Then, $M_i$, being an implementation of $\mathscr{C}_i$ for $i = 1, 2$, satisfies:

for every state $(q_i, q_{M_i})$ that is reachable in $E_{\mathscr{C}_i} \times M_i$, we have:

\[
\Sigma_i^{\text{out}} \cap \Sigma_{M_i}(q_{M_i}) \subseteq \Sigma_i^{\text{out}} \cap \text{may}_i(q_i)
\]  \hspace{1cm} (8.73)

\[
\Sigma_i^{\text{out}} \cap \text{must}_i(q_i) \subseteq \Sigma_i^{\text{out}} \cap \Sigma_{M_i}(q_{M_i})
\]

On the other hand, (8.70) amounts to proving

for every state $(q, q_{M_1}, q_{M_2})$ of states that is reachable in $E_{\mathscr{C}} \times M_1 \times M_2$, the following holds:

\[
\Sigma^{\text{out}} \cap \Sigma_{M_1 \times M_2}(q_{M_1}, q_{M_2}) \subseteq \Sigma^{\text{out}} \cap \text{may}_{\mathscr{C}}(q)
\]  \hspace{1cm} (8.74)

\[
\Sigma^{\text{out}} \cap \text{must}_{\mathscr{C}}(q) \subseteq \Sigma^{\text{out}} \cap \Sigma_{M_1 \times M_2}(q_{M_1}, q_{M_2})
\]

whereas (8.71) amounts to proving

for every state $(q_E, q_{M_2}, q_1)$ that is reachable in $E \times M_2 \times \mathscr{C}_{1}^{\text{may}}$, we have: $\Sigma^{\text{in}} \cap \Sigma_{E \times M_2}(q_E, q_{M_2}) \subseteq \text{must}_1(q_1)$

\hspace{1cm} (8.75)

Proof that (8.72, 8.73) imply (8.74): We first consider (8.74) in the case in which $q = \Pi_{12}$, the universal state. By the definition of $\Pi_{12}$, (8.74) is trivially satisfied in this case.

We next consider (8.74) in the case $q \neq \Pi_{12}$, meaning that $q = (q_1, q_2) \in Q_1 \times Q_2$ was not an exception state of the pre-composition $\mathscr{C} = \text{def} \mathscr{C}_1 \otimes \mathscr{C}_2$, see (8.68). In this case we have

\[
\text{may}_{\mathscr{C}}(q) = \text{may}_1(q_1) \cap \text{may}_2(q_2)
\]

\[
\text{must}_{\mathscr{C}}(q) = \text{must}_1(q_1) \cap \text{must}_2(q_2)
\]  \hspace{1cm} (8.76)

Thus, in the case $q \neq \Pi$, (8.74) rewrites

\[
\Sigma^{\text{out}} \cap \Sigma_{M_1}(q_{M_1}) \cap \Sigma_{M_2}(q_{M_2}) \subseteq \Sigma^{\text{out}} \cap \text{may}_1(q_1) \cap \text{may}_2(q_2)
\]

\[
\Sigma^{\text{out}} \cap \text{must}_1(q_1) \cap \text{must}_2(q_2) \subseteq \Sigma^{\text{out}} \cap \Sigma_{M_1}(q_{M_1}) \cap \Sigma_{M_2}(q_{M_2})
\]  \hspace{1cm} (8.77)

On the other hand, we assume (8.73). Taking the union with $\Sigma_j^{\text{out}}$, $j \neq i$, of each side of the two inclusions of (8.73) yields, for $i = 1, 2$:

\[
\Sigma^{\text{out}} \cap \Sigma_{M_i}(q_{M_i}) \subseteq \Sigma^{\text{out}} \cap \text{may}_i(q_i)
\]

\[
\Sigma^{\text{out}} \cap \text{must}_i(q_i) \subseteq \Sigma^{\text{out}} \cap \Sigma_{M_i}(q_{M_i})
\]

which implies (8.77), i.e., equivalently, (8.74).

Proof that (8.72, 8.73) imply (8.75): Consider first the case where $q = \Pi_{12}$ in (8.72), which, by the definition of $\Pi_{12}$, implies $\Sigma^{\text{in}} \cap \Sigma_E(q_E) = \emptyset$. Then, (8.75) is trivially satisfied since $\Sigma^{\text{in}} \cap \Sigma_{E \times M_2}(q_E, q_{M_2}) = \Sigma^{\text{in}} \cap \Sigma_E(q_E) \cap \Sigma_{M_2}(q_{M_2}) = \emptyset$. 

Proof that (8.72, 8.73) imply (8.74): We first consider (8.74) in the case in which $q = \Pi_{12}$, the universal state. By the definition of $\Pi_{12}$, (8.74) is trivially satisfied in this case.
We thus focus on the case \( q \neq \top \) in (8.72), meaning that \( q = (q_1, q_2) \in Q_1 \times Q_2 \) was not an exception state of the pre-composition \( \mathcal{C} \overset{\text{def}}{=} \mathcal{C}_1 \otimes \mathcal{C}_2 \), see (8.68). Thus, (8.76) holds in this case. We now prove (8.75) by contradiction. Suppose (8.75) is falsified, i.e.,

\[
\exists \alpha \in \Sigma^{\text{in}} : \begin{cases}
\alpha \in \Sigma_E(q_E) \\
\alpha \in \Sigma_{M_2}(q_{M_2}) \\
\alpha \notin \text{must}_1(q_1)
\end{cases}
\]  

(8.78)

By (8.76) we get \( \alpha \notin \text{must}_E(q) \), which, by (8.72), implies \( \alpha \notin \Sigma_E(q_E) \), which contradicts (8.78). This finishes the proof of (8.69).

It thus remains to prove that

\( \mathcal{C} \) is minimal in \( \mathcal{C}_{12} \) for refinement order. (8.79)

If not, there would exist \( \mathcal{C}' \in \mathcal{C}_{12}, \mathcal{C}' \preceq \mathcal{C}, \mathcal{C}' \neq \mathcal{C} \), meaning that

\( \mathcal{C}' \) satisfies (4.2)  

(8.80)

\( E_{\mathcal{C}'} \supseteq E_{\mathcal{C}} \) and \( M_{\mathcal{C}'} \subseteq M_{\mathcal{C}} \)  

(8.81)

either \( E_{\mathcal{C}'} \supseteq E_{\mathcal{C}} \) or \( M_{\mathcal{C}'} \subset M_{\mathcal{C}} \)  

(8.82)

Case \( E_{\mathcal{C}'} \supseteq E_{\mathcal{C}} \) in (8.82): In this case \( E_{\mathcal{C}'} \) simulates \( E_{\mathcal{C}} \) but the converse is not true. By Lemma 8.9, there exists a word \( w \in \Sigma^* \) leading environments \( E_{\mathcal{C}'} \) and \( E_{\mathcal{C}} \) to states \( q' \) and \( q \) for which

\[
\Sigma_{E_{\mathcal{C}'}}(q') \supseteq \Sigma_{E_{\mathcal{C}}}(q)
\]  

(8.83)

Observe that (8.83) requires that \( q \neq \top \), the top state of components. Due to Condition 2 of Definition 8.7 and step (a) of the construction of the maximal environment in the same definition, (8.83) requires the existence of some action

\[
\alpha \in \Sigma_{E_{\mathcal{C}'}}(q'), \alpha \notin \Sigma_{E_{\mathcal{C}}}(q), \alpha \in \Sigma_{\text{out}}
\]

which is a contradiction since, being a component, the environment \( E_{\mathcal{C}} \) is receptive, meaning that \( \Sigma_{E_{\mathcal{C}}}(q) \supseteq \Sigma_{\text{out}} \) for any \( q \). Thus we must have \( E_{\mathcal{C}'} = E_{\mathcal{C}} \). As a consequence,

\[
E_{\mathcal{C}'} = E_{\mathcal{C}}
\]  

(8.84)
8.3. Modal Interfaces with fixed alphabet

Case $\mathcal{E}_{\mathcal{C}}' = \mathcal{E}_{\mathcal{C}}$ and $\mathcal{M}_{\mathcal{C}}' \subset \mathcal{M}_{\mathcal{C}}$ in (8.82): By the characterization (4.2) of the contract composition in the meta-theory, $\mathcal{M}_{\mathcal{C}}'$ must contain all compositions $M_1 \times M_2$, for $M_1$ and $M_2$ ranging over the sets of all implementations of $\mathcal{C}_1$ and $\mathcal{C}_2$. On the other hand, by the characterization (8.74) and using (8.84), checking the implementation relation $M \models ^= \mathcal{C}'$ requires inspecting what happens at reachable states $(q, q_M)$ of the composition $E_{\mathcal{C}} \times M$. Due to the composition with the maximal environment $E_{\mathcal{C}}$, exception states $q$ are never accessed — see (8.66) regarding exception states. In the pruning operation (8.68), tentative transitions to such states are replaced by transitions to the universal state $\top_{12}$. On the other hand, at a reachable state $(\top_{12}, q_M)$ of the composition $E_{\mathcal{C}} \times M$, ready sets of $M$ are unconstrained. To summarize:

- $\mathcal{M}_{\mathcal{C}}'$ must contain all compositions $M_1 \times M_2$, for $M_1$ and $M_2$ ranging over the sets of all implementations of $\mathcal{C}_1$ and $\mathcal{C}_2$;

- $\mathcal{M}_{\mathcal{C}}'$ must contain every component $M$ such that, at every state $(q, q_M)$ that is reachable in $E_{\mathcal{C}} \times M$ and satisfies $q \neq \top_{12}$, we have:

$$\Sigma^{\text{out}} \cap \Sigma_M(q_M) \subseteq \Sigma^{\text{out}} \cap \text{may}(q)$$
$$\Sigma^{\text{out}} \cap \text{must}(q) \subseteq \Sigma^{\text{out}} \cap \Sigma_M(q_M)$$

But these two properties exactly characterize the set $\mathcal{M}_{\mathcal{C}}$ of all implementations of $\mathcal{C}$. Thus, $\mathcal{M}_{\mathcal{C}}' \supseteq \mathcal{M}_{\mathcal{C}}$ holds, a contradiction. This finishes the proof of (8.79), and the theorem is proved. $\square$

Regarding associativity: Unfortunately, Axiom 4 of the meta-theory does not hold for Modal Interfaces. The obstruction originates from the post-processing (8.68) for pruning illegal pairs of states in the composition, thus resulting in an abstraction (the converse of a refinement) of the pre-composition. Hence, we cannot apply Theorem 4.6 of the meta-theory to derive the associativity of contract composition. Figure 8.1 shows a counterexample.

Indeed, Gerald Lüttgen, Walter Vogler and co-workers [62, 63] found errors in [227] where the parallel composition was erroneously claimed to be associative, with similar errors for interface automata reported in [66]. Counterexamples were provided for both cases. In Section 8.4 we present the
approach of [62, 63] and show that our \textit{n}-ary parallel composition coincides with the parallel composition of this reference.

### 8.3.6 Quotient

As a preliminary step, a \textit{pre-quotient} operator is defined. It was first introduced in [227], where it is called “quotient”. It is called pre-quotient in this monograph because it does not coincide with the notion of quotient introduced in the meta-theory — it actually is the adjoint of the pre-composition introduced in Definition 8.10. As the pre-quotient does not address compatibility properly, we then address compatibility by introducing a notion of \textit{compatible quotient}. Unfortunately, we were not able to prove that the latter is the adjoint of the parallel composition and we guess it is not — this is why we do not call it “quotient”. Finding a true quotient matching the meta-theory is still open. In [62, 63] Bujtor et al. propose a mild modification of our operators here, ensuring associativity of the parallel composition and making the quotient the adjoint of the parallel composition. We develop in Section 8.4 a comparison of the two approaches for the parallel composition.

**Definition 8.12 (pre-quotient).** The \textit{pre-quotient} \(C_1/C_2\) of two Modal Interfaces \(C_1\) and \(C_2\) is only defined if \(\Sigma_{1}^{in} \cap \Sigma_{2}^{out} = \emptyset\) and is defined according to the following three steps procedure.

![Figure 8.1: Counterexample to the satisfaction of Axiom 4 for Modal Interfaces.](image)
1. Consider first the Modal Interface

\[ C_1 \circ C_2 = (\Sigma^\text{in}, \Sigma^\text{out}, Q, \top, q_0, \to, \cdot \to), \]

where:

(a) \( \Sigma^\text{out} = \Sigma_1^\text{out} \setminus \Sigma_2^\text{out} \), \( Q = (Q_1 \times Q_2) \cup \{ \top, \top \} \);

(b) The initial state \( q_0 \) is defined as follows:

\[
q_0 = \begin{cases} 
\top & \text{if } q_{2,0} \notin Q_2 \\
(q_{1,0}, q_{2,0}) & \text{otherwise}
\end{cases}
\]

(c) The \textit{may} and \textit{must} transition relations are the least relations satisfying the following axioms:

\[ (Q1) \quad (q_1, q_2) \overset{\alpha}{\rightarrow} (q'_1, q'_2) \quad \text{if} \quad q_1 \overset{\alpha_1}{\rightarrow} q'_1 \text{ and } q_2 \overset{\alpha_2}{\rightarrow} q'_2 \]

\[ (Q2) \quad (q_1, q_2) \overset{\alpha}{\rightarrow} (q'_1, q'_2) \quad \text{if} \quad q_1 \overset{\alpha_1}{\rightarrow} q'_1 \text{ and } q_2 \overset{\alpha_2}{\rightarrow} q'_2 \]

\[ (Q3) \quad (q_1, q_2) \overset{\alpha}{\rightarrow} (q'_1, q'_2) \quad \text{if} \quad q_1 \overset{\alpha_1}{\rightarrow} q'_1 \text{ and } \alpha \notin \text{must}_2(q_2) \]

\[ (Q4) \quad (q_1, q_2) \overset{\alpha}{\rightarrow} (q'_1, q'_2) \quad \text{if} \quad q_1 \overset{\alpha_1}{\rightarrow} q'_1 \text{ and } q_2 \overset{\alpha_2}{\rightarrow} q'_2 \text{ and } \alpha \notin \text{must}_1(q_1) \]

\[ (Q5) \quad (q_1, q_2) \overset{\alpha}{\rightarrow} \top \quad \text{if} \quad \alpha \notin \text{must}_1(q_1) \cup \text{may}_2(q_2) \]

\[ (Q6) \quad \top \overset{\alpha}{\rightarrow} \top \]

\[ (Q7) \quad \top \overset{\alpha}{\rightarrow} \top \]

2. States of the form \((\top, q_2)\) are replaced by the incompatible state \(\top\).

3. Applying rule \((Q3)\) results in an inconsistent state, thus, we apply the reduction \((8.55)\). \(\square\)

Observe that, by construction, the \textit{must} and \textit{may} sets of the modal interface \(C_1 \circ C_2\) defined in the first step of Definition 8.12 are given by the following equalities, for all state pairs \((q_1, q_2) \in Q_1 \times Q_2\):

\[
\text{must}(q_1, q_2) = \text{must}_1(q_1)
\]

\[
\text{may}(q_1, q_2) = \left[ \text{must}_1(q_1) \cap \text{must}_2(q_2) \right] \cup \left[ \text{may}_1(q_1) \cap \neg \text{must}_1(q_1) \right] \cup \left[ \neg \text{may}_1(q_1) \cup \text{may}_2(q_2) \right]
\]

This definition is justified by the following result [227]:

**Lemma 8.13.** \(C \otimes C_2 \leq C_1\) if and only if \(C \leq C_1 \circ C_2\). \(\square\)
Proof. Given two modal interfaces $\mathcal{C}_1$ and $\mathcal{C}_2$ on the same alphabet $\Sigma$ and such that $\Sigma_{\text{out}}^1 \subseteq \Sigma_{\text{out}}^2$, construct the modal interface $\mathcal{C} = \mathcal{C}_1 / \mathcal{C}_2$ according to the rules of Definition 8.12. Consider a Modal Interface $\mathcal{C}_3$ such that $\Sigma_{\text{out}}^3 = \Sigma_{\text{out}}^1 \setminus \Sigma_{\text{out}}^2$ and set $\mathcal{C}' = \mathcal{C}_3 \otimes \mathcal{C}_2$.

Proof of $\mathcal{C}_3 \leq \mathcal{C} \Rightarrow \mathcal{C}' \leq \mathcal{C}_1$: Assume $\mathcal{C}_3 \leq \mathcal{C}$. Consider the particular case where $\mathcal{C}_3$ is inconsistent ($q_{3,0} \not\in Q_3$). Then $\mathcal{C}'$ is also inconsistent and $\mathcal{C}' \leq \mathcal{C}_1$. The other particular case is when $q_0 = \top$. This happens only when $\mathcal{C}_2$ is inconsistent, which implies that $\mathcal{C}'$ is inconsistent and $\mathcal{C}' \leq \mathcal{C}_1$. Consider the general case $q_{3,0} \in Q_3$ and $q_0 \neq \top$. Assume that $\rho \subseteq Q_3 \times ((Q_1 \times Q_2) \cup \{\top, \top\})$ is the refinement relation, with $(q_{3,0}, q_{1,0}, q_{2,0}) \in \rho$. Let us prove that $\rho' = \{(q_3, q_2, q_1) \in Q_3 \times Q_2 \times Q_1 \mid (q_3, q_1, q_2) \in \rho\}$ is a refinement relation between $\mathcal{C}'$ and $Q_1$.

For every $(q_3, q_2, q_1) \in \rho'$ and every $\alpha \in \Sigma$, if $\alpha \in \text{may}_3(q) \cap \text{may}_2(q_2) \cap \text{may}_1(q_1)$, then, there exist unique $q_3', q_2'$ and $q_1'$ such that $q_3 \xrightarrow{\alpha} q_3', q_2 \xrightarrow{\alpha} q_2'$ and $q_1 \xrightarrow{\alpha} q_1'$. Since $(q_3, q_1, q_2) \in \rho$ and $\rho$ is a refinement relation, state $(q_1, q_2)$ of $\mathcal{C}$ is not inconsistent. Consequently, Axiom (Q3) does not apply in this state, meaning that either $\alpha \notin \text{must}_1(q_1)$ or $\alpha \in \text{must}_1(q_1) \cap \text{must}_2(q_2)$. In the first case, Axiom (Q4) applies, while in the second case, Axiom (Q1) applies. Both axioms enable a $(q_1, q_2) \xrightarrow{\alpha} (q_1', q_2')$ transition in $\mathcal{C}$. Therefore, $(q_3', q_1', q_2') \in \rho$ and $(q_3', q_2', q_1') \in \rho'$. Now, for every $(q_3, q_1, q_2) \in \rho$, using equations (8.86), we have:

\[
\begin{align*}
\{ \text{must}_3(q_3) \supset \text{must}_1(q_1) \\
\text{may}_3(q_3) \subseteq [\text{must}_1(q_1) \cap \text{must}_2(q_2)] \cup \neg [\text{may}_1(q_1) \cap \neg \text{must}_1(q_1)] \cup [\text{may}_1(q_1) \cup \text{may}_2(q_2)]
\end{align*}
\]

The may and must functions of $\mathcal{C}'$ satisfy the following equations:

\[
\begin{align*}
\{ \text{must}'(q_3, q_2) &= \text{must}_3(q_3) \cap \text{must}_2(q_2) \\
\text{may}'(q_3, q_2) &= \text{may}_3(q_3) \cap \text{may}_2(q_2)
\end{align*}
\]

Every state pair $(q_1, q_2)$ of $\mathcal{C}$ such that $\text{must}_1(q_1) \not\subseteq \text{must}_2(q_2)$ is made inconsistent. Therefore, every $(q_3, q_1, q_2) \in \rho$ is such that $\text{must}_1(q_1) \not\subseteq \text{must}_2(q_2)$.
8.3. Modal Interfaces with fixed alphabet

263

This implies must0 (q3 , q2 ) ⊇ must1 (q1 ). Now, with the may functions:


may0 (q3 , q2 ) ⊆ must1 (q1 ) ∩ must2 (q2 ) ∩ may2 (q2 )
∪


may (q ) ∩ ¬must1 (q1 ) ∩ may2 (q2 )
∪
  1 1


¬ may1 (q1 ) ∪ may2 (q2 ) ∩ may2 (q2 )


= may1 (q1 ) ∩ ¬must1 (q1 ) ∩ may2 (q2 )
∪


must1 (q1 ) ∩ must2 (q2 )
⊆ may1 (q1 ) ,

which proves may0 (q3 , q2 ) ⊆ may1 (q1 ). This concludes the proof that ρ0 is a
refinement relation. Since (q3,0 , q1,0 , q2,0 ) ∈ ρ, (q3,0 , q2,0 , q1,0 ) ∈ ρ0 holds and
C 0  C1 . This completes the proof of the implication C3  C ⇒ C 0  C1 .

Proof of the converse implication C 0  C1 ⇒ C3  C : Assume C 0  C1 .
Consider the particular case where C 0 is inconsistent. By definition of the
pre-composition, this implies that either C3 or C2 are inconsistent. C3 inconsistent implies C3  C . C2 inconsistent implies C incompatible and
C3  C . The other particular case is C1 incompatible, in which case C is
also incompatible and C3  C holds. Consider now the general case: Assume ρ ⊆ Q3 × Q2 × Q1 is the refinement relation, with (q3,0 , q2,0 , q1,0 ) ∈ ρ.
Set ρ0 = {(q3 , q1 , q2 )|(q3 , q2 , q1 ) ∈ ρ} ∪ (Q3 × {>}) and let us prove that it is a
refinement relation.
For every (q3 , q1 , q2 ) ∈ ρ0 , and every α ∈ may3 (q3 ) ∩ may(q1 , q2 ), since
(q3 , q2 , q1 ) ∈ ρ, the following inclusion on must sets holds:
must3 (q3 ) ∩ must2 (q2 ) ⊇ must1 (q1 ) = must(q1 , q2 )
Therefore, must3 (q3 ) ⊇ must(q1 , q2 ). Regarding the may sets:
may3 (q3 ) ∩ ¬may(q1 , q2 ) =


may (q ) ∩ ¬must1 (q1 ) ∪ ¬must2 (q2 ) ∩
 3 3
 

¬may1 (q1 ) ∪ must1 (q1 ) ∩ may1 (q1 ) ∪ may2 (q2 ) =


may3 (q3 ) ∩ ¬must1 (q1 ) ∩ ¬may1 (q1 ) ∩ may1 (q1 ) ∪


may3 (q3 ) ∩ ¬must1 (q1 ) ∩ ¬may1 (q1 ) ∩ may2 (q2 ) ∪



may3 (q3 ) ∩ ¬must1 (q1 ) ∩ must1 (q1 ) ∩ may1 (q1 ) ∪


may3 (q3 ) ∩ ¬must1 (q1 ) ∩ must1 (q1 ) ∩ may2 (q2 ) ∪


may3 (q3 ) ∩ ¬must2 (q2 ) ∩ ¬may1 (q1 ) ∩ may1 (q1 ) ∪


may3 (q3 ) ∩ ¬must2 (q2 ) ∩ ¬may1 (q1 ) ∩ may2 (q2 ) ∪





may3 (q3 ) ∩ ¬must2 (q2 ) ∩ must1 (q1 ) ∩ may1 (q1 ) ∪


may3 (q3 ) ∩ ¬must2 (q2 ) ∩ must1 (q1 ) ∩ may2 (q2 ) = ∅

—

using equation (8.86)

—
—
—
—
—
—
—
—
—
—
—
—
—
—

expanding into
disjunctive normal form
empty clause
empty, since may3 (q3 )∩
may2 (q2 ) ⊆ may1 (q1 )
empty clause
empty clause
empty clause
empty, since may3 (q3 )∩
may2 (q2 ) ⊆ may1 (q1 )
empty, since
must1 (q1 ) ⊆ must2 (q2 )
empty, since
must1 (q1 ) ⊆ must2 (q2 )


This proves that $\text{may}_3(q_3) \subseteq \text{may}(q_1, q_2)$. Now, let us prove that for every $(q_3, q_1, q_2) \in \rho'$, and every $\alpha \in \text{may}_3(q_3)$, the unique successor state $(q'_3, q')$ such that $q \overset{\alpha}{\rightarrow} q'_3$ and $(q_1, q_2) \overset{\alpha}{\rightarrow} (q')$ belongs to $\rho'$. Two cases must be distinguished:

1. Case $\alpha \in (\text{must}_1(q_1) \cap \text{must}_2(q_2)) \cup (\text{may}_1(q_1) \setminus \text{must}_1(q_1)) \cap \text{may}_2(q_2)$, where rules (Q1) or (Q4) apply and $q' = (q'_1, q'_2)$ such that $q \overset{\alpha}{\rightarrow} q'_1$ and $q \overset{\alpha}{\rightarrow} q'_2$. Therefore $(q'_3, q'_1, q'_2) \in \rho$ and, by definition, $(q'_3, q'_1, q'_2) \in \rho'$.

2. Case $\alpha \notin \text{must}_1(q) \cup \text{may}_2(q_2)$, where rule (Q5) applies and $q' = \top$. By definition $(q'_3, \top) \in \rho'$. Finally, every $q_3 \in Q_3$ refines the $\top$ state of $\mathcal{C}$. This completes the proof of $\mathcal{C}_3 \preceq \mathcal{C}$.

The problem with the pre-quotient operator is that the interface $\mathcal{C} = \mathcal{C}_1 / \mathcal{C}_2$ may be incompatible with $\mathcal{C}_2$, in a pessimistic sense — meaning that $\mathcal{C} \bowtie \mathcal{C}_2$ may have reachable illegal states. A compatible quotient operation $\mathcal{C}' = \mathcal{C}_1 / \mathcal{C}_2$ is now introduced, guaranteeing that $\mathcal{C}' \bowtie \mathcal{C}_2$ has no reachable illegal state. To define the compatible quotient operator, we need to define the peer of a modal interface $\mathcal{C}$, which is the greatest modal interface $\mathcal{C}'$ on a given I/O alphabet that is compatible in a pessimistic sense with $\mathcal{C}$.

**Definition 8.13 (Peer).** Given a modal interface $\mathcal{C}_1$, given input/output alphabets $\Sigma_2^{\text{in}}$ and $\Sigma_2^{\text{out}}$ such that $\Sigma_2^{\text{out}} \subseteq \Sigma_1^{\text{in}}$, define the modal interface $\mathcal{C}_2 = \text{Peer}_{\Sigma_2^{\text{in}}, \Sigma_2^{\text{out}}}(\mathcal{C}_1)$, where:

$$q_{2,0} = \begin{cases} q_{1,0} & \text{if } \top \neq q_{1,0} \in Q_1 \\ \top & \text{if } q_{1,0} \notin Q_1 \\ \bot & \text{if } q_{1,0} = \top \end{cases}$$

Transition relations $\rightarrow_2$ and $\rightarrow_2$ are the least relations satisfying the axioms

(P1) $\alpha \overset{\alpha}{\rightarrow}_2 q'$ if $\alpha \overset{\alpha}{\rightarrow}_1 q'$ and $\alpha \in \Sigma_2^{\text{out}}$

(P2) $\alpha \overset{\alpha}{\rightarrow}_2 q'$ if $\alpha \overset{\alpha}{\rightarrow}_1 q'$ and $\alpha \in \Sigma_2^{\text{out}}$

(P3) $\alpha \overset{\alpha}{\rightarrow}_2 \top$ if $\alpha \notin \text{may}_1(q)$ and $\alpha \in \Sigma_2^{\text{out}}$

(P4) $\alpha \overset{\alpha}{\rightarrow}_2 q'$ if $\alpha \overset{\alpha}{\rightarrow}_1 q'$ and $\alpha \in \Sigma_2^{\text{out}}$

(P5) $\alpha \overset{\alpha}{\rightarrow}_2 q'$ if $\alpha \overset{\alpha}{\rightarrow}_1 q'$ and $\alpha \in \Sigma_1^{\text{in}} \cap \Sigma_2^{\text{in}}$

(P6) $\alpha \overset{\alpha}{\rightarrow}_2 \top$ if $\alpha \notin \text{may}_1(q)$ and $\alpha \in \Sigma_1^{\text{in}} \cap \Sigma_2^{\text{in}}$

(P7) $\top \overset{\alpha}{\rightarrow}_2 \top$

$\square$
Remark that
\[ Q_2 = Q_1 \cup \{ \top \} \text{ and } \bot \notin Q_2 \] (8.87)

and that for all state \( q \in Q_1 \), the following equalities hold:
\[
\begin{align*}
\{ \text{must}_2(q) \} &= \text{may}_1(q) \cap \Sigma_1^{\text{out}} \\
\{ \text{may}_2(q) \} &= \Sigma_1^{\text{out}} \cup (\Sigma_1^{\text{in}} \cap \Sigma_2^{\text{in}}) \cup (\text{must}_1(q) \cap \Sigma_2^{\text{out}})
\end{align*}
\] (8.88)

The following lemma states that \( \text{Peer}_{\Sigma_1^{\text{in}}, \Sigma_2^{\text{out}}}^{\text{in}(C_1)} \) characterizes the set of interfaces that are compatible, in a pessimistic sense, to \( C_1 \).

**Lemma 8.14.** Given an interface \( C_1 \), and input/output alphabets \( \Sigma_2^{\text{in}}, \Sigma_2^{\text{out}} \) satisfying the assumption of Definition 8.13, for all \( C \) on input/output alphabets \( \Sigma_2^{\text{in}}, \Sigma_2^{\text{out}} ; C \otimes C_1 \) possesses no reachable illegal state if and only if \( C \leq \text{Peer}_{\Sigma_1^{\text{in}}, \Sigma_2^{\text{out}}}^{\text{out}(C_1)} \). \( \square \)

**Proof.** Throughout the proof we call \( C_2 \) the peer \( \text{Peer}_{\Sigma_1^{\text{in}}, \Sigma_2^{\text{out}}}^{\text{out}(C_1)} \).

**If statement:** Assume \( C \leq C_2 \), with a refinement relation \( \rho \subseteq Q \times Q_2 \) and recall (8.87). To prove the if statement, we prove that

(i) \( C \otimes C_1 \) has no reachable illegal state and

(ii) the set of reachable states of \( C \otimes C_1 \) is included into \( \rho' = \rho \cap (Q \times Q_1) \).

We begin with (i). For every \( (q, q_2) \in \rho' \), since state \( q \) refines state \( q_2 \), the following inclusions hold:
\[
\begin{align*}
\{ \text{must}(q) \} &\supseteq \text{must}_2(q_2) \\
\{ \text{may}(q) \} &\subseteq \text{may}_2(q_2)
\end{align*}
\]

Using equations (8.88) with \( q_2 \) substituted for \( q \), these inclusions imply:
\[
\begin{align*}
\{ \text{must}(q) \} &\supseteq \text{may}_1(q_2) \cap \Sigma_1^{\text{out}} \\
\{ \text{may}(q) \} &\subseteq \text{must}_1(q_2) \cap \Sigma_2^{\text{out}}
\end{align*}
\]

These two inclusions ensure that every output may transition of \( C_1 \) in state \( q_2 \) is matched by a must input transition of \( C \) in state \( q \), and conversely, every output may transition of \( C \) in state \( q \) is matched by an input must transition of \( C_1 \) in state \( q_2 \). Hence, no state pair \( (q, q_2) \in \rho' \) is illegal.
We next move to (ii) and consider the following three cases, depending on the initial state $q_0$ of $C$.

**Case $q_0 < Q$:** In this case $C$ is inconsistent, hence, so is $C \otimes C_1$, hence its set of reachable states is empty. Thus, (ii) is vacuously true in this case.

**Case $q_0 = \top$:** In this case, $C$ is incompatible, hence so is $C_2$. Hence, $q_0 = 0$, that is, $Q_1 = \emptyset$, which implies $Q \times Q_1 = \emptyset$, so that (ii) is also vacuously true in this case.

**Other cases:** We shall prove by induction that all reachable states of $C \otimes C_1$ belong to $\rho'$:

- **Initial state** $(q_0, q_{1,0}) \in \rho'$.
  - For every state $(q, q_1) \in \rho'$, let us prove that every successor state $(q', q'_1)$ that can be reached from $(q, q_1)$ by a may transition of $C \otimes C_1$ also belongs to $\rho'$. For every $\alpha \in \text{may}(q) \cap \text{may}_1(q_1)$, consider the unique $(q', q'_1)$ such that $q \xrightarrow{\alpha} q'$ and $q_1 \xrightarrow{\alpha} q'_1$. Since $(q, q_1)$ is not an illegal state of $C \otimes C_1$, $\text{may}(q) \cap \Sigma_{\text{out}}^2 \subseteq \text{must}_1(q_1)$. Hence, using equalities (8.88), $\alpha \in \text{may}(q) \cap \text{may}_2(q_1)$. Therefore, $(q', q'_1) \in \rho'$.

This finishes the proof of the if statement.

**Only-if statement:** Assume that $C \otimes C_1$ has no reachable illegal state. Let us denote $\rho \subseteq Q \times Q_1$ the set of reachable states of $C \otimes C_1$. Set $\rho' = \rho \cup (Q \times \{\top\})$ and prove that $\rho'$ is a refinement relation.

For every $(q, q_1) \in \rho$, we first prove that $(q, q_1)$ satisfy the axioms of a refinement relation. Note that $(q, q_1)$ is not an illegal state of $C \otimes C_1$. Hence:

\[
\begin{align*}
\text{must}(q) & \supseteq \text{may}_1(q_1) \cap \Sigma_{\text{out}}^1 \\
\text{must}_1(q_1) & \supseteq \text{may}(q) \cap \Sigma_{\text{out}}^2
\end{align*}
\]

Using equations (8.88), the above inclusion imply the following inclusions, which are the first two axioms of modal refinement:

\[
\begin{align*}
\text{must}(q) & \supseteq \text{must}_2(q_1) \\
\text{may}(q) & \subseteq \text{may}_2(q_1)
\end{align*}
\]

Next, we have to prove the third and last axiom of modal refinement, namely that states that the successors $(q', q'_1)$ of state pair $(q, q_1) \in \rho$ for
8.3. Modal Interfaces with fixed alphabet

The may transition relations of both \( C \) and \( C_2 \) belong to \( \rho' \). For every \( \alpha \in \text{may}(q) \cap \text{may}(q_1) \), there exists a unique state pair \((q', q'_1)\) such that \( q \xrightarrow{\alpha} q' \) and \( q_1 \xrightarrow{\alpha} q'_1 \). Five cases shall be considered where \((P1) - (P6)\) refer to the axioms introduced in Definition 8.13:

1. If \( \alpha \in \Sigma^{\text{out}}_2 \), since \((q, q_1)\) is not an illegal state of \( C \otimes C_1 \), \( \alpha \in \text{must}_1(q_1) \). \((P4)\) is the only axiom that applies hence \( q_1 \xrightarrow{\alpha} q'_1 \). Therefore \((q', q'_1) \in \rho \).

2. If \( \alpha \in \Sigma^{\text{out}}_1 \cap \text{may}_1(q_1) \), then \((P2)\) is the only axiom that justifies the transition \( q_1 \xrightarrow{\alpha} q'_1 \) and consequently \( q \xrightarrow{\alpha} q'_1 \). This implies that \((q', q'_1) \in \rho \).

3. If \( \alpha \in \Sigma^{\text{out}}_1 \setminus \text{may}_1(q_1) \), then \((P3)\) applies and \( q'_1 = \top \). Hence \((q', q'_1) \in \rho' \).

4. If \( \alpha \in \Sigma^{\text{in}}_1 \cap \Sigma^{\text{in}}_2 \cap \text{may}_1(q_1) \), then \((P5)\) applies and \((q', q'_1) \in \rho \).

5. If \( \alpha \in (\Sigma^{\text{in}}_1 \cap \Sigma^{\text{in}}_2) \setminus \text{may}_1(q_1) \), then \((P6)\) applies and \( q'_1 = \top \). Hence \((q', q'_1) \in \rho' \).

Finally, we observe that, for every state \( q \) of \( C \), \( q \) refines state \( \top \) of \( C_2 \) since \( \top \) is universal and \( q \) differs from \( \top \). This finishes the proof that \( \rho' \) is a refinement relation and, \( C \preceq C_2 \). \( \square \)

Using the peer operator, the compatible quotient operator may be defined as follows:

**Definition 8.14.** The **compatible quotient** \( C_1 / C_2 \) of two Modal Interfaces \( C_1 \) and \( C_2 \) is defined as follows:

\[
C_1 / C_2 = (C_1 / C_2) \wedge \text{Peer}_{\Sigma^{\text{in}}_1 \Sigma^{\text{out}}_2}(C_2)
\]

Where \( \Sigma^{\text{in}} = \Sigma^{\text{in}}_1 \cup \Sigma^{\text{out}}_2 \) and \( \Sigma^{\text{out}} = \Sigma^{\text{out}}_1 \cap \Sigma^{\text{in}}_2 \). \( \square \)

The following theorem characterizes the compatible quotient. Unfortunately, this characterization does not correspond to the adjoint of the parallel composition, as the meta-theory would require.
Theorem 8.15. Whenever it is defined, the compatible quotient $C_1 / C_2$ is characterized by:

$$C_1 / C_2 = \max \left\{ C \middle| \begin{array}{l} C \otimes C_2 \text{ possesses no reachable illegal state, and} \\ C_1 \otimes C_2 \preceq C \end{array} \right\}$$

and $C_1 / C_2 \otimes C_2 \preceq C_1$ holds. □

Proof. This theorem is an immediate consequence of Lemmas 8.13 and 8.14. Observe that, for $C_1 \otimes C_2$ possessing no reachable illegal state, $C_1 \otimes C_2 = C_1$ holds. □

8.4 The approach by Gerald Lüttgen, Walter Vogler et al.

In this section we reproduce the small subset of the approach of [62] (see also [66] for an extended version) dealing with parallel composition. Non-determinism is also considered in this reference, but we do not develop that aspect here. Whenever needed, to distinguish this approach from the one we developed before, we add the prefix “LV-”.

Definition 8.15 ([62, 66]). The pre-composition $C_1 \otimes^\text{LV} C_2$ of two Modal Interfaces is only defined if $\Sigma_1^\text{out} \cap \Sigma_2^\text{out} = \emptyset$ and is the Modal Interface given by:

$\Sigma^\text{out} = \Sigma_1^\text{out} \cup \Sigma_2^\text{out}$, $Q = (Q_1 \times Q_2) \cup \{ \top \}$, $q_0 = (q_{1,0}, q_{2,0})$, $\top$, and the two transition relations:

$$(q_1, q_2) \overset{\alpha}{\to} (q_1', q_2') \iff q_i \overset{\alpha}{\to} q_i', i = 1, 2$$

$$(q_1, q_2) \overset{\cdot}{\to} (q_1', q_2') \iff q_i \overset{\cdot}{\to} q_i', i = 1, 2$$

By construction, the must and may sets of $C_1 \otimes^\text{LV} C_2$ are given by:

$$\text{must}(q_1, q_2) = \text{must}_1(q_1) \cap \text{must}_2(q_2)$$

$$\text{may}(q_1, q_2) = \text{may}_1(q_1) \cap \text{may}_2(q_2)$$

Say that a state $(q_1, q_2)$ of $C_1 \otimes^\text{LV} C_2$ is illegal if

$$\text{may}_1(q_1) \cap \Sigma_1^\text{in} \not\subseteq \text{must}_2(q_2)$$

or

$$\text{may}_2(q_2) \cap \Sigma_2^\text{in} \not\subseteq \text{must}_1(q_1)$$

(8.90)
8.4. The approach by Gerald Lüttgen, Walter Vogler et al.

When the pre-composition is in an illegal state, an implementation of one of the interfaces may output an action that is not accepted by an implementation of the other interface — this situation would contradict the objective of independent implementation. Hence we expect a valid environment to prevent the implementation from reaching an illegal state.

Define the set $E$ of exception states as being the smallest set of states $(q_1, q_2) \in Q_1 \times Q_2$ satisfying the following properties, of which the first one has been added compared to (8.66):

- if either $q_1 = \top_1$ or $q_2 = \top_2$, then $(q_1, q_2) \in E$;
- if $q$ is illegal, then $q \in E$;
- if $q \xrightarrow{\alpha} q'$ holds for some $\alpha \in \Sigma^{\text{out}}$ and $q' \in E$, then $q \in E$.

(8.91)

Exception states are pruned away from $C_1 \otimes C_2$ by using again (8.67, 8.68).

**Definition 8.16.** Definition 8.15 completed with the post-processing (8.67, 8.68) yields the contract composition $C_1 \otimes^{lv} C_2$, which no longer possesses illegal states.

Recall that it is shown in [62] that the parallel composition of Definition 8.16 is associative. Indeed, we can easily recover this result as a consequence of the meta-theory:

**Theorem 8.16.** The parallel composition $\otimes^{lv}$ satisfies Axiom 4 of the meta-theory, thus showing that $\otimes^{lv}$ is associative.

**Proof.** We invoke Theorem 4.7 with $\sqsubseteq$ being the simulation relation (8.5) for i/o-automata. We have that $m_C$ and $M_C$ are equal to $C_{\text{must}}$ and $C_{\text{may}}$, made receptive. It remains to show that, thanks to the reinforced pruning introduced in [62], we have

$$(C_1 \otimes^{lv} C_2)^{\text{must}} = C_1^{\text{must}} \times C_2^{\text{must}}$$

(8.92)

This follows directly from the definition (8.91) of the exception states and the post-processing (8.67, 8.68) applied to them. It was essential that states of the form $(q_1, q_2)$ with $q_1 = \top_1$ or $q_2 = \top_2$ were included in the set of exception states. Failure of doing this in the original approach of [227] caused the failure of $\otimes$ to be associative.
To complete the clarification, the following result states that $\otimes^{LV}$ is the “associative extension” of our $n$-ary parallel composition of Definition 8.11:

**Theorem 8.17.** Let $\mathcal{E}_i, i \in I$ be a set of Modal Interfaces following Definition 8.7. We can regard them as Modal Interfaces in the LV-sense by having no transition leading to the universal state $\top$. Then: $\otimes_{i \in I}^{LV} \mathcal{E}_i = \otimes_{i \in I} \mathcal{E}_i$. □

**Proof.** The only difference between the two parallel compositions lies in the definitions (8.66) and (8.91) of the exception states. For this case, however, the first item of (8.91) is vacuous. So, for this case, the two definitions simply coincide. □

### 8.5 Modal Interfaces with variable alphabet

As a general principle, every relation or operator introduced in Section 8.3 (for Modal Interfaces with a fixed alphabet $\Sigma$) is extended to the case of variable alphabets by 1) extending and equalizing alphabets, and then 2) applying the relations or operators of Section 8.3 to the resulting Modal Interfaces. For all frameworks we studied so far, alphabet extension was performed using inverse projections, see Section 5.2. For instance, this is the procedure used in defining the composition of i/o-automata: extending alphabets in i/o-automata is by adding, at each state and for each added action, a self-loop labeled with this action. The very reason for using this mechanism is that it is neutral for the composition in the following sense: it leaves the companion i/o-automaton free to perform any wanted local action.

So, for Modal Interfaces, what would be a neutral procedure for extending alphabets? Indeed, considering (8.62) or (8.64) yields two different answers, namely:

- for (8.62) : $\{ \alpha \in \text{may}^1(q_1) \text{ and } \alpha \in \text{whatever}^2(q_2) \} \Rightarrow \alpha \in \text{whatever}(q_1, q_2)$
- for (8.64) : $\{ \alpha \in \text{must}^1(q_1) \text{ and } \alpha \in \text{whatever}^2(q_2) \} \Rightarrow \alpha \in \text{whatever}(q_1, q_2)$

where “whatever” denotes either may or must. Consequently, neutral alphabet extension is by adding may self-loops for the conjunction, and must self-loops for the composition.
The bottom line is that we need different extension procedures. These observations explain why alphabet extension is properly handled neither by Interface Automata (see the last paragraph of Section 8.2) nor by A/G contracts (see the end of Section 5.2). These theories do not offer enough flexibility for ensuring neutral extension for all relations or operators. We now list how alphabet extension must be performed for each relation or operator, for two Modal Interfaces $\mathcal{C}_1$ and $\mathcal{C}_2$ (the reader is referred to [227] for justifications).

Throughout this section, and, more generally, when alphabet extensions are considered, every alphabet comes with its partitioning $\Sigma = \Sigma^\text{in} \cup \Sigma^\text{out}$ and $\Sigma' \supseteq \Sigma$ means $\Sigma^\text{in} \supseteq \Sigma^\text{in}$ and $\Sigma^\text{out} \supseteq \Sigma^\text{out}$.

With this in mind, we define the strong extension of $\mathcal{C}$ to $\Sigma' \supseteq \Sigma$, written $\mathcal{C}^{\Sigma'}$, as the modal interface $\mathcal{C}^{\Sigma'} = (\Sigma^\text{in}, \Sigma^\text{out}, Q, q_0, \rightarrow', \leftarrow')$, where:

\[
\rightarrow' = \rightarrow \cup \{ (q, \alpha, q) \mid q \in Q \text{ and } \alpha \in \Sigma' \setminus \Sigma \}
\]

\[
\leftarrow' = \leftarrow \cup \{ (q, \alpha, q) \mid q \in Q \text{ and } \alpha \in \Sigma' \setminus \Sigma \}
\] (8.93)

Similarly, we define the weak extension of $\mathcal{C}$ to $\Sigma' \supseteq \Sigma$, written $\mathcal{C}^{\Sigma'}$, as the modal interface

\[
\mathcal{C}^{\Sigma'} = (\Sigma^\text{in}, \Sigma^\text{out}, Q, q_0, \rightarrow, \leftarrow')
\] (8.94)

where $\rightarrow'$ is defined as in (8.93) while $\rightarrow$ is kept unchanged. In words, only may self-loops are added in weak extensions, whereas both may and must self-loops are added in the strong extension.

Observe that the strong extension uses the classical inverse projection everywhere. The weak extension, however, proceeds differently with the must transitions in that it forbids the legal environments to submit additional actions as its outputs.

Using weak and strong alphabet equalization, the relations and operations introduced in Section 8.3 extend to variable alphabets as indicated now. In the following theorem, $(\Sigma, \Sigma')$ is a pair such that $\Sigma^\text{in} \subseteq \Sigma'^\text{in}$ and $\Sigma^\text{out} \subseteq \Sigma'^\text{out}$, and $(\Sigma_1, \Sigma_2, \Sigma)$ denotes a triple such that $\Sigma = \Sigma_1 \cup \Sigma_2$. Contract $\mathcal{C}$ has alphabet $\Sigma$, and contract $\mathcal{C}_i$ has alphabet $\Sigma_i$. Finally, for each listed operation, decomposition $\Sigma_i = \Sigma^\text{in}_i \sqcup \Sigma^\text{out}_i$ is such that composability conditions are satisfied:
Theorem 8.18. The following relations and operators:

\[ M' \models^u \mathcal{C} \quad ::= \quad M' \models^u \mathcal{C}^{\Sigma'} \]
\[ E' \models^e \mathcal{C} \quad ::= \quad E' \models^e \mathcal{C}^{\Sigma'} \]
\[ \mathcal{C}' \preceq \mathcal{C} \quad ::= \quad \mathcal{C}' \preceq \mathcal{C}^{\Sigma'} \]
\[ \mathcal{C}_1 \land \mathcal{C}_2 \quad ::= \quad \mathcal{C}_1^{\Sigma} \land \mathcal{C}_2^{\Sigma} \]
\[ \mathcal{C}_1 \otimes \mathcal{C}_2 \quad ::= \quad \mathcal{C}_1^{\Sigma} \otimes \mathcal{C}_2^{\Sigma} \]
\[ \mathcal{C}_1 \sqcup \mathcal{C}_2 \quad ::= \quad \mathcal{C}_1^{\Sigma} \sqcup \mathcal{C}_2^{\Sigma} \]

instantiate the meta-theory, with the exception of the compatible quotient \( \mathcal{C}_1 / \mathcal{C}_2 \).

\[ \square \]

Proof. The first two formulas just provide definitions, so no proof is needed for them. Their purpose is to characterize the weakly and strongly extended Modal Interfaces in terms of their sets of environments and implementations. For both extensions, allowed output actions of the implementations are augmented whereas mandatory actions are not. For the weak extension, legal environments are not modified in that no additional output action is allowed for them. In contrast, for the strong extension, legal environments are allowed to submit any additional output action. These observations justify the other formulas.

\[ \square \]

In (8.95) the condition \( \Sigma' \supseteq \Sigma \) is required when defining the refinement. Using the same notations as in Theorem 8.18, we can remove this condition on alphabets by stating

\[ \mathcal{C}_2 \preceq \mathcal{C}_1 \quad ::= \quad \mathcal{C}_2^{\Sigma} \preceq \mathcal{C}_1 \land \mathcal{C}_2 \]

(8.96)

We will freely use (8.96) in the sequel.

8.6 Decomposing a contract as a composition of subcontracts

A difficult step in the management of contracts was illustrated in Figure 2.2 of Section 2.1. It consists in decomposing a contract \( \mathcal{C} \) into a composition of subcontracts satisfying

\[ \bigotimes_{i \in I} \mathcal{C}_i \preceq \mathcal{C} \]

(8.97)
8.6. Decomposing a contract as a composition of subcontracts

input: $\mathcal{C}, \Sigma^{in}, \Sigma^{out}, \Sigma'^{in}, \Sigma'^{out}$; output: $\mathcal{C}'$

let rest($X$) =

while $X$ has not been visited,
do
1. mark $X$ visited
2. for every $\alpha \in \Sigma'$ do
   2.1 let $Y = \text{next}(\alpha, X)$
   2.2 let $Z = \varepsilon$-closure($\Sigma - \Sigma', Y$)
   2.3 let $m = \text{Op}(m_{\mathcal{C}}(q, \alpha) \mid q \in X)$, where $\text{Op}$ is defined in (8.100)
   2.4 add to $\mathcal{C}'$ a transition $(X, \alpha, Z)$ with modality $m$
2.5 rest($Z$)
done

let restrict($\mathcal{C}$) =
1. let $X_0 = \varepsilon$-closure($\Sigma - \Sigma', q_0$)
2. set initial state of $\mathcal{C}'$ to $X_0$
3. rest($X_0$)
4. return $\mathcal{C}'$ and reduce it

$\alpha : \{ \quad \quad \quad \}$ maps to $X \xrightarrow{\alpha} Z$

$\varepsilon : \quad \quad \quad \}$

Figure 8.2: Algorithm for computing the restriction $\mathcal{C}_{\Sigma'}$. The function $m_{\mathcal{C}}(q, \alpha)$ with codomain [cannot, may, must, inconsistent] returns the modality of action $\alpha$ at state $q$ for $\mathcal{C}$. The figure on the bottom illustrates the function “rest”. For $X$ a set of states of $\mathcal{C}$, define $Y$ as being the image of $X$ under all possible $\alpha$-transitions originating from some state belonging to $X$. Then, define $Z$ as being the $\varepsilon$-closure of $Y$ when hiding actions not belonging to $\Sigma'$, i.e., the set of states that can be reached from $Y$ through a sequence of $\text{may}$-transitions labeled with an action not belonging to $\Sigma$. $X \xrightarrow{\alpha} Z$ is created as a transition of $\mathcal{C}'$ depicted as a double arrow, with the modality $m \in \{ \text{cannot, may, must} \}$ computed using the function $\text{Op}$. 
where subcontract $C_i$ has alphabet $\Sigma_i = \Sigma^\text{in}_i \cup \Sigma^\text{out}_i$. As a prerequisite to (8.97), the designer has to guess some topological architecture by decomposing the alphabet of actions of $C$ as

$$\Sigma = \bigcup_{i \in I} \Sigma_i \quad , \quad \Sigma_i = \Sigma^\text{in}_i \cup \Sigma^\text{out}_i$$

(8.98)
such that composability conditions regarding inputs and outputs hold. Guessing architectural decomposition (8.98) relies on the designer’s understanding of the system and how it should naturally decompose — this typically is the world of SysML.

Finding decomposition (8.97) is, however, technically difficult in that it involves behaviors [185, 182]. It is particularly difficult if $C$ itself is a conjunction of viewpoints or requirements, which typically occurs in requirements engineering, see Chapter 10:

$$C = \bigwedge_{k \in K} C_k$$

(8.99)
The typical approach in moving from (8.99) to (8.97) is by guessing the result and then verifying the correctness of the guess. In this section we go beyond this by proposing a synthesis method, from (8.99) to (8.97). This is the approach followed in the Parking Garage example of Chapter 10.

Let $C$ be a Modal Interface with alphabet $\Sigma = \Sigma^\text{in} \cup \Sigma^\text{out}$ and let $(\Sigma^\text{in}, \Sigma^\text{out})$ be two input and output sub-alphabets such that $\Sigma^\text{in} \subseteq \Sigma^\text{in}$ and $\Sigma^\text{out} \subseteq \Sigma^\text{out}$. Set $\Sigma' = \Sigma^\text{in} \cup \Sigma^\text{out}$ and define the restriction of $C$ to $\Sigma'$, denoted by $C^\downarrow_{\Sigma'}$, which is computed using the procedure shown in Figure 8.2, where the operator Op on modalities is defined as follows, where $M = \{m \in \Sigma | q \in X\}$:

$$\alpha \in \Sigma^\text{in} : Op(M) = \text{if } [\text{must} \in M \text{ or } \text{may} \in M] \text{ then } \text{must} \text{ else can}\not$$

$$\alpha \in \Sigma^\text{out} : Op(M) = \bigwedge M, \text{ for the order: inconsistent } < \text{must } < \text{cannot } < \text{may }$$

(8.100)

Observe that the states of the restriction correspond to sets of states of the original Modal Interface.

**Lemma 8.19.** Assume that $C$ and $C' = \text{def } C^\downarrow_{\Sigma'}$ possess no inconsistent state. Then:

- $C \parallel C'$ introduced in (8.85) possesses no inconsistent state;  
- decomposition $C' \otimes (C \parallel C') \leq C$ holds;  
- the pair $(C', C \parallel C')$ has no incompatible pair of states;
8.6. Decomposing a contract as a composition of subcontracts

and $C_{\Sigma'}$ is maximal (with respect to the refinement order) having these properties.

Proof. We first prove (8.101). The only case in which a reachable state $(q, X)$ of $C'/C_{\Sigma'}$ is inconsistent is when $m_C(q, \beta) = \text{must}$ whereas $m_{C_{\Sigma'}}(X, \beta) \neq \text{must}$. But the operator $Op$ used in instruction 2.3 of the algorithm of Figure 8.2 prevents this situation from occurring. Actually, the choice of $Op$ makes $C'$ maximal (for refinement order) having property (8.101). Properties (8.102) and (8.103) follow from Theorem 8.15 about the compatible quotient.

Comment 8.20. Why should one bother with computing the restriction $C_{\Sigma'}$? Why not simply choose an arbitrary Modal Interface $C'$ with alphabet $\Sigma'$ and then taking the decomposition $C' \otimes (C'/C')$? Indeed, Theorem 8.15 tells us that $C' \otimes (C'/C') \preceq C$ would anyway be guaranteed. Our motivations for nevertheless introducing the restriction operator are the following:

- We want to avoid that step 3 of Definition 8.12 prunes some states of $C_{\Sigma'}$, since we think that everything from our guess $C'$ should survive. This motivates requiring (8.101).

- We want $C'$ to be a guess of how to “project” $C$ on subalphabet $\Sigma'$. In doing so, we hope that this guess will capture as much as possible from the original Modal Interface $C$. This motivates our request for maximality.

Decomposition (8.102) can be used as follows when subcontracting:

Algorithm 1. We are given some system-level contract $C$. The top-level designer guesses some topological architecture following (8.98) and recursively decomposes:

$$C = C_0 \succeq C_{0|\Sigma_1} \otimes C_1 \succeq C_{0|\Sigma_1} \otimes C_{1|\Sigma_2} \otimes C_2 \succeq \ldots \succeq C_{0|\Sigma_1} \otimes \ldots \otimes C_{n-1|\Sigma_i} \overset{\text{def}}{=} C(\Sigma_1) \otimes \ldots \otimes C(\Sigma_n)$$ (8.104)
Provided that the $C(\Sigma _i), i = 1, \ldots , n,$ are all consistent, decomposition (8.104) yields a refinement of $C$ by a composition of subcontracts involving no pair of illegal states.

It is the duty of the designer to find an architectural decomposition (8.98) ensuring that the $C(\Sigma _i), i = 1, \ldots , n,$ are all consistent. A good choice is to select this decomposition so that interactions between the different subalphabets are “weak” in some informal sense. We offer no formal guideline for this.

### 8.7 Modal interfaces as Assume / Guarantee contracts

In this section we explain how to represent, using Modal Interfaces, Assume / Guarantee contracts of the form $C = ([A_1, \ldots A_n], G)$, where the assumptions $A_i$ and the guarantee $G$ are Modal Interfaces. Regarding the i/o status of the assumptions and the guarantee, the following holds:

- The guarantee $G$ specifies the expected behavior of a component. We assume that its i/o alphabet is $\Sigma _G = \Sigma _{in}^G \cup \Sigma _{out}^G$.
- Assumptions $A_i$ adopt the conjugate point of view, since they specify expected properties of the environment. Hence, for $i = 1 \ldots n$, the i/o alphabet $\Sigma _{A_i} = \Sigma _{in}^{A_i} \cup \Sigma _{out}^{A_i}$ of assumption $A_i$, should be such that:
  \[
  \Sigma _{in}^{A_i} \cap \Sigma _{out}^{A_j} = \emptyset \quad \text{for all} \quad j = 1 \ldots n
  \]
  \[
  \Sigma _{in}^{A_i} \cap \Sigma _{in}^{G} = \emptyset \quad \text{and} \quad \Sigma _{out}^{A_i} \cap \Sigma _{out}^{G} = \emptyset
  \]

The next question is: How several assumptions shall be combined together? How do guarantees and assumptions interact?

#### 8.7.1 A vending machine example

These questions are first answered in the context of a simple example: a vending machine serving tea or coffee. This is an academic example distributed as part of the MICA tool [70], an implementation of the Modal Interface theory, supporting contract-based reasoning. This example relates the design of a system with three input actions ?coin, ?tea_req, and ?coffee_req, and two output actions !tea and !coffee. Question and exclamation marks
Modal interfaces as Assume/Guarantee contracts

are only a reminder of the i/o status of the action: ! stands for output, and ? is for input. Assumptions, for this particular example, have no may transitions. The reason is twofold: The environment has no control on the output actions of the vending machine, hence ?tea and ?coffee transitions have the modality must. Regarding the actions under control of the environment, !coin, !tea_req and !coffee_req, the most permissive environment is considered, which explains that these transitions also have a must modality.

The behavior of the vending machine is specified as a set of assumptions and guarantees, and a set of contracts relating the previously defined assumptions/guarantees. The assumptions defined in Figures 8.3 state that the user is expected to insert not more than one coin (assumption $A_1$) and press more than one button (assumption $A_2$) per transaction. The expected behavior of the vending machine is also specified in a modular way, with the modal interfaces in Figure 8.4. These specifications will be used as guarantees, to be paired with assumptions. Guarantee $G_1$ states that the machine shall not deliver any beverage before having received payment. Guarantee $G_2$ expresses that the machine can not deliver tea when coffee has been requested and deliver coffee when tea has been requested. Guarantee $G_3$ simply states that the

![Figure 8.3](image1.png)

**Figure 8.3:** On the left, assumption $A_1$, “users shall not insert more than one coin per transaction”. On the right, assumption $A_2$, “users shall not press more than one button per transaction”. In both modal specification the initial state is labeled 0 and has a losange shape.

![Figure 8.4](image2.png)

**Figure 8.4:** On the left, guarantee $G_1$, “the machine shall not deliver any beverage before having received payment”. In the middle, guarantee $G_2$, “the machine can not deliver tea when coffee has been requested and deliver coffee when tea has been requested”. On the right, guarantee $G_3$, “the machine must be receptive to input actions”.


vending machine must be receptive to its input actions, meaning that it cannot refuse \(?\text{coin}, \?\text{tea req}, \?\text{coffee req}\).

Three contracts are considered: \(\mathcal{C}_i = ([A_1, A_2], G_i), i = 1 \ldots 3\). Contract \(\mathcal{C}_i\) states that \(G_i\) must hold, under the assumption that both \(A_1\) and \(A_2\) hold. We capture this by stating that assumptions compose using the conjunction operator, to form a global assumption shown in Figure 8.5, on the top left:

\[
A = \text{def } A_1 \land A_2.
\]

Consider guarantee \(G_1\). Although their alphabets are compatible, specifications \(G_1\) and \(A\) have different alphabets, and, thus, the issue of alphabet equalization must be considered. It turns out that alphabet equalization is not performed in the same way for guarantees and assumptions.

- Regarding assumptions:

\[
\begin{align*}
\Sigma_{A}^{\text{in}} & = \Sigma_{A}^{\text{in}} \cup \Sigma_{G}^{\text{out}} \\
\Sigma_{A}^{\text{out}} & = \Sigma_{A}^{\text{out}} \cup \Sigma_{G}^{\text{in}}
\end{align*}
\]

*Strong equalization is used on assumptions,* meaning that self-loop transitions with a *must* modality are inserted in every state of the assumption and for every action in the alphabet of the guarantee that is missing in the alphabet of the assumption. Remark that, for the particular instance of assumption \(A\) in our vending machine example, no
equalization needs to be performed, and \( A' = A \) as a result of equalization.

- Regarding guarantees:

\[
\Sigma_G'_{\text{in}} = \Sigma_G^{\text{in}} \cup \Sigma_A^{\text{out}} \\
\Sigma_G'_{\text{out}} = \Sigma_G^{\text{out}} \cup \Sigma_A^{\text{in}}
\]

The same operation applies to the guarantee, using however weak equalization, where may self-loops are inserted. The rationale for using weak equalization on guarantees is simply that the equalized guarantee \( G'_1 \) should be neutral to actions that are not observable by the guarantee.

The equalized guarantee \( G'_1 \) and assumption \( A' \) are composable and their composition

\[
G_1 \bowtie A = \text{def} \quad G'_1 \otimes A' \quad (8.105)
\]

(see Figure 8.5, top right) is the guarantee \( G_1 \) put in the context of assumption \( A \). It specifies the set of compositions of a system satisfying the guarantee \( G_1 \), with an environment satisfying assumption \( A \). The contract is then computed by relieving \( G_1 \bowtie A \) of the assumption \( A \). This is defined using the compatible quotient operator, shown at the bottom of Figure 8.5:

\[
\mathcal{C}_1 = \text{def} \quad (G_1 \bowtie A)/A \quad (8.106)
\]

Contracts \( \mathcal{C}_2 = (G_2 \bowtie A)/A \) and \( \mathcal{C}_3 = (G_3 \bowtie A)/A \) are defined in the same way and are shown the top and middle of Figure 8.6. The global contract is the conjunction of the three contracts \( \mathcal{C} = \text{def} \quad \mathcal{C}_1 \wedge \mathcal{C}_2 \wedge \mathcal{C}_3 \), shown at the bottom of Figure 8.6. This contract is clearly incomplete, since it allows implementations of the vending machine that output !tea or !coffee without any request ?tea_req or ?coffee_req. Completing the specification of the vending machine with a fourth contract is an easy exercise, left to the reader.

### 8.7.2 Comparison with A/G contracts of Chapter 5

Modal A/G contracts developed in Section 8.7.1 bear many similarities with the A/G contracts detailed in Chapter 5. In this section we show how A/G contracts can be mapped to Modal A/G contracts in such a way that some, but not all properties of the A/G contract algebra are preserved. Before doing this, we observe the following discrepancies, which prevent a perfect matching:
Figure 8.6: Top, contract $C_2$. Middle, contract $C_3$. Bottom, global contract $C = C_1 \land C_2 \land C_3$

- The A/G contracts of Chapter 5 are oblivious to i/o orientation, while in Modal Interfaces, actions are either an output or an input, and this plays an important role in the theory.

- A second difference is that A/G contracts are based on a dataflow or synchronous semantics, where behavior is defined as streams of values, one per variable, or as a sequence of partial assignments of the variables. This differs from Modal Interfaces, where behavior is defined as sequences of actions taken in a finite alphabet. So far the above two discrepancies can be seen to be technical. The next one, however, is more fundamental.
• In A/G contracts \( C = (A, G) \), the assumption \( A \) is handled in a rigid way: \( E \models C' \) amounts to \( E \subseteq A \), and, as a consequence, \( C' \preceq C \) requires \( A' \supseteq A \). In contrast, formula \( C' = (G \rhd A)/A \) does not define \( A \) uniquely and, thus, refinement cannot constrain \( A \) directly.

These discrepancies explain why the two theories cannot perfectly match, and one can only hope for a partial embedding of A/G contracts in the Modal Interface theory. This is detailed below. In this development, we use the subscripts \( \text{AG} \) and \( \text{MI} \) to distinguish contract relations or operations according to the A/G contract and Modal Interface frameworks.

We are given a finite alphabet \( \Sigma \) and we consider the A/G contract \((A, G)\), where \( A \) and \( G \) are non-empty prefix-closed regular subsets of \( \Sigma^* \). A and G are the languages of deterministic finite transition systems, which we also denote by \( A = (\Sigma, Q^A, q_0^A, \rightarrow^A) \) and \( G = (\Sigma, Q^G, q_0^G, \rightarrow^G) \). Assumptions and guarantees are mapped to Modal Interfaces:

\[
A^m = (\Sigma^{\text{in}} = \emptyset, \Sigma^{\text{out}} = \Sigma, Q^A, q_0^A, \rightarrow^A, \rightarrow^A) \\
G^m = (\Sigma^{\text{in}} = \emptyset, \Sigma^{\text{out}} = \Sigma, Q^G, q_0^G, \emptyset, \rightarrow^G)
\]

meaning that assumptions are mapped to rigid interfaces, where all transitions are \textit{must} transitions and all actions are outputs, whereas guarantees are mapped to relaxed interfaces with output actions and only \textit{may} transitions.

Recall that Modal A/G contracts are given by:

\[
C = (G^m \rhd A^m)/A^m
\]

The so defined mapping \((A, G) \rightarrow C\) preserves refinement:

**Theorem 8.21.** \( (A_1, G_1) \preceq_{\text{AG}} (A_2, G_2) \) implies \( C_1 \preceq_{\text{MI}} C_2 \). □

The converse implication does not hold in general, as shown by the counter-example of Figure 8.7. The reason is that A/G contract refinement requires that assumptions \( A_1 \) and \( A_2 \) are comparable, while they are not directly related by the Modal Interface refinement relation.

**Proof.** We use notations from Chapter 5. For \( M \) a modal interface, \( M^1 \) denotes the interface \( M \) with input and output actions interchanged, see (8.37).

---

5With reference to Chapter 5 and particularly formula (5.1), we consider sets of behaviors for a singleton variable \( v \) whose domain is \( D_v = \Sigma \).
Denote by $M_i$ the language of the maximal implementations of $\mathcal{C}_i$. Observe that $G_i \bowtie A_i$ has an empty must transition relation. Therefore, the must transition relation of $\mathcal{C}_i$ is also empty. Therefore $\mathcal{C}_1 \preceq_{\text{MI}} \mathcal{C}_2$ reduces to $M_1 \subseteq M_2$.

Denote by $G'_i = (G_i \cup \neg A_i)^\downarrow$ the saturated guarantee. Recall that $G'_i$ is the largest prefix-closed language contained in $G_i \cup \neg A_i$. By construction, the language of the may transition relation of $G'_i \bowtie A'_i$ is $G_i \cap A_i$. Therefore the language of the may transition relation of $\mathcal{C}_i$ is equal to $(G_i \cap A_i) \cup \neg A_i)\downarrow = G'_i$.

Hence $\mathcal{C}_1 \preceq_{\text{MI}} \mathcal{C}_2$ iff $G'_1 \subseteq G'_2$, which concludes the proof.

Regarding contract composition, the following theorem states that the Modal Interface image of the composition of two contracts $(A_i, G_i)$, $i = 1, 2$ is equivalent to the composition of the images of the two contracts. Define the images of the three A/G contracts as Modal Interfaces:

$$
\begin{align*}
(A_0, G_0) &= (A_1, G_1) \otimes_{\text{AG}} (A_2, G_2), \\
\mathcal{C}_i &= (G_i^{\text{m}} \bowtie A_i^{\text{m}})/A_i^{\text{m}} \text{ for } i = 0, 1, 2
\end{align*}
$$

For $\mathcal{C} = (\Sigma^{\text{in}}, \Sigma^{\text{out}}, Q, q_0, \rightarrow, \cdots)$ a Modal Interface, we define its dual $\overline{\mathcal{C}}$ obtained by exchanging, in $\mathcal{C}$, the input/output status.

**Theorem 8.22.** Modal Interfaces $\mathcal{C}_0$ and $\mathcal{C}_1 \otimes_{\text{MI}} \mathcal{C}_2$ refine one-another.

**Proof.** Modal Interfaces $\mathcal{C}_i$, $i = 0 \ldots 2$ are consistent and their must transition relations are empty. Denote by $M_i$ the maximal implementation of $\mathcal{C}_i$. Using the same reasoning as in the previous proof, $\mathcal{C}_0 \equiv \overline{\mathcal{C}_1} \otimes \mathcal{C}_2$ reduces to $M_0 = \mathcal{C}_0 \equiv \overline{\mathcal{C}_1} \otimes \mathcal{C}_2$.
By definition of contract composition $G_0 = G'_1 \cap G'_2$ and $A_0 = \max\{A | A = A^1, A \cap G'_2 \subseteq A_1, A \cap G'_1 \subseteq A_2\}$, where $G'_i = (G_i \cup \neg A_i)^\downarrow$, $i = 1, 2$ are the saturated guarantees. By construction, $M_i = (G_i \cup \neg A_i)^\downarrow$.

Using the fact that $G_0$ is saturated, the definition of $M_0$ expands to: $M_0 = (G_1 \cup \neg A_1)^\downarrow \cap (G_2 \cup \neg A_2)^\downarrow = M_1 \cap M_2$. \hfill $\Box$

## 8.8 Bibliographical note

As explained in [105, 82, 177, 118, 226, 227], Interface Theories make no explicit distinction between assumptions and guarantees. These notions are implicitly supported, however, through the particular semantics of these models.

### 8.8.1 Interface Automata, variants and extensions

Interface Automata were proposed by de Alfaro, Henzinger and co-workers in a remarkable series of papers [106, 105, 82, 83, 118] as a candidate theory of interfaces. In these references, Interface Automata focused primarily on parallel composition and compatibility. Quoting from [105]: "Two interfaces can be composed and are compatible if there is at least one environment where they can work together". The idea is that the resulting composition exposes as an interface the needed information to ensure that incompatible pairs of states cannot be reached. This can be achieved by using the possibility, for a component, to refuse selected inputs from the environment at a given state [105, 82]. In contrast to our development in Section 8.2, no sets of environments and implementations are formally associated to an Interface Automaton in the original developments of the concept. A refinement relation for Interface Automata was defined in [105]—with the same definition as ours—it could not, however, be expressed in terms of sets of implementations. Properties of interfaces are described in game-based logics, e.g., ATL [13], with a theoretical high-cost complexity. The original semantics of an Interface Automaton was given by a two-player game between: an input player that represents the environment (the moves are the input actions), and an output player that represents the component itself (the moves are the output actions). It is interesting to compare, in particular, the so-called “asynchronous” setting developed in [106, 105] to the “synchronous” setting
of [82]. In the former, the parallel composition is made difficult due to the consideration of illegal pairs of states and the pruning that follows. In contrast, the “synchronous” setting of [82] characterizes the interface through a pair of synchronous transition relations constraining inputs and outputs, respectively. The parallel composition proposed in this reference is rather reminiscent of the one proposed for A/G contracts in this monograph in point 3 of Definition 5.4. Finally, the recent work [64, 66] revisits the foundations of Interface Automata following the so-called “asynchronous” setting of [105]. In particular, a counterexample is given, showing that the original parallel composition of [105] is not associative and corrections to the framework are provided.

In [118], the framework of Synchronous Interfaces was enriched with a notion of conjunction (called shared refinement). This development was further elaborated in [111] for the topic of time-triggered scheduling. Synchronous Relational Interfaces [250, 251] have been proposed to capture functional relations between the inputs and the outputs associated to a component. More precisely, input/output relations between variables are expressed as first-order logic formulas over the input and output variables. Two types of composition are then considered, connection and feedback. Given two relational interfaces \( C_1 \) and \( C_2 \), the first one consists in connecting some of the output variables of \( C_1 \) to some of the input variables of \( C_2 \) whereas feedback composition allows one to connect an output variable of an interface to one of its own inputs. The developed theory supports refinement, compatibility and also conjunction. The recent work [154] studies conditions that need to be imposed on interface models in order to enforce independent implementability with respect to conjunction.

An algebraic theory of interface automata was recently proposed in [86] following a line similar to [87]. Specifications (called “components” in that reference) are characterized by two sets of observable and inconsistent prefix-closed abstract sets of traces constituting the interface of the specification. Refinement, conjunction and disjunction, parallel composition, and quotient, are provided, thus offering a comprehensive framework.

Sociable Interfaces [104] combine the approach presented in the previous paragraph with interface automata [105, 106] by enabling communication via
shared variables and actions. First, the same action can appear as a label of both input and output transitions. Secondly, global variables do not belong to any specific interface and can thus be updated by multiple interfaces. Consequently, communication and synchronization can be one-to-one, one-to-many, many-to-one, and many-to-many. Symbolic algorithms for checking the compatibility and refinement of sociable interfaces have been implemented in TICC [7]. Software Interfaces were proposed in [81], as a pushdown extension of interface automata (which are finite state). Pushdown interfaces are needed to model call-return stacks of possibly recursive software components. This paper contains also a comprehensive interface description of Tiny OS, an operating system for sensor networks. Moore machines and related reactive synchronous formalisms are very well suited to embedded systems modeling. Extending interface theories to a reactive synchronous semantics is therefore meaningful. Several contributions have been made in this direction, starting with Moore and Bidirectional Interfaces [82]. In Moore Interfaces, each variable is either an input or an output, and this status does not change in time. Bidirectional Interfaces offer added flexibility by allowing variables to change I/O status, depending on the local state of the interface. Communication by shared variable is thus supported and, for instance, allows distributed protocols or shared buses to be modeled. In both formalisms, two interfaces are deemed compatible whenever no variable is an output of both interfaces at the same time, and every legal valuation of the output variables of one interface satisfies the input predicate of the other. The main result of the paper is that parallel composition of compatible interfaces is monotonic with respect to refinement. Note that Moore and Bidirectional Interfaces force a delay of at least one transition between causally dependent input and output variables, exactly like Moore machines. Reference [79] develops the concept of simulation distances for interfaces, thereby taking robustness issues into account by tolerating errors. Finally, Web services Interfaces were proposed in [52].

---

6 This formalism is thus not purely synchronous and is mentioned in this section with a slight abuse.
7 https://github.com/tinyos/tinyos-main
8.8.2 Modal Interfaces, variants and extensions

Properties expressed as sets of traces can only specify what is forbidden. Unless time is explicitly invoked in such properties, it is not possible to express mandatory behaviors for designs. Modalities were proposed by Kim Larsen [180, 19, 61] as a simple and elegant framework to express both allowed and mandatory properties. Modal Specifications basically consist in assigning a modality \textit{may} or \textit{must} to each possible transition of a system. They have been first studied in a process-algebraic context [180, 175] in order to allow for loose specifications of systems. Since then, they have been considered for automata [178] and formal languages [224, 225] and applied to a wide range of application domains (see [19] for a complete survey). Informally, a \textit{must} transition is available in every component that realizes the modal specification, while a \textit{may} transition needs not be. A modal specification thus represents a set of \textit{models}—unfortunately, models of modal transition systems are often call “implementations” in the literature, which is unfortunate in our context. We prefer keeping the term “model” and reserve the term “implementation” for the entities introduced in Sections 8.2 and 8.3. Modal Specifications offer built-in conjunction of specifications [179, 228]. The expressiveness of Modal Specifications has been characterized as a strict fragment of the Hennessy-Milner logic in [61] and also as a strict fragment of the \textit{mu}-calculus in [130]. The formalism is rich enough to specify safety properties as well as restricted forms of liveness properties. Modal Interfaces with a correct notion of compatibility were introduced in [226, 227] and the problem of alphabet equalization with weak and strong alphabet extensions was first correctly addressed in the same references. In [32], compatibility notions for Modal Interfaces with the passing of internal actions are defined. Contrary to the approach reviewed before, a pessimistic view of compatibility is followed in [32], i.e., two Modal Interfaces are only compatible if incompatibility between two interfaces cannot occur in any environment. A verification tool called MIO Workbench is available. The quotient of Modal Specifications was studied in [181, 225]. The work by Bujtor et al. [62, 63] is particularly important. It points out a remaining pitfall in [227], namely the parallel composition for Modal Interfaces of [227] fails to be associative. [62, 63] exhibits a counterexample and proposes an adaptation of the Modal Interfaces of [227] where the parallel composition becomes associative. This
modification introduces the universal state $\top$ and makes an extensive use of it in keeping track of so-called exception states. Our present writing adopts much of the approach of [62, 63].

In addition to providing the above correction, [62, 63] addresses nondeterminism in a very elegant way. Actually, determinism plays a role in the modal theory. Non-deterministic Modal Interfaces have possibly non-deterministic i/o-automata as class of components. Their corresponding computational procedures are of higher complexity than for deterministic ones. A Modal Interface is said to be deterministic if its may-transition relation is deterministic. For nondeterministic Modal Interfaces, modal refinement is incomplete [178]: there are nondeterministic Modal Interfaces $C_1$ and $C_2$ for which the set of implementations of $C_1$ is included in that of $C_2$ without $C_1$ being a modal refinement of $C_2$. Hence refinement according to the metatheory is not exactly instantiated but only approximated in a sound way. A decision procedure for implementation inclusion of nondeterministic Modal Interfaces does exist but turns out to be \text{exptime}-complete [20, 35] whereas the problem is \text{ptime}-complete if determinism is assumed [228, 36]. The benefits of the determinism assumption in terms of complexity for various decision problems on modal specifications is underlined in [36]. With the aim to preserve deadlock freedom, [65] defines a new refinement relation for modal transition systems (MTS). This refinement “supports itself” e.g. in the sense of thoroughness – in contrast to the standard modal refinement. Finally, the longstanding conflict between unspecified inputs being allowed in Interface Automata but forbidden in MTS is solved in [62, 63]. The “merge” of nondeterministic Modal Specifications regarded as partial models has been considered in [252]. This operation consists in looking for common refinements of initial specifications and is thus similar to the conjunction operation presented here. In [252, 131], algorithms to compute the maximal common refinements (which are not unique when non-determinism is allowed) are proposed. They are implemented in the tool MTSA [117]. Assume/guarantee contracts viewed as pairs of Modal Specifications were proposed in [136]. It thus combines the flexibility offered by the clean separation between assumptions and guarantees and the benefits of a modal framework. Several operations are then studied: refinement, parallel composition, conjunction and priority of aspects. This last operation composes aspects in a hierarchical order,
such that in case of inconsistency, an aspects of higher priority overrides a lower-priority contract. The synthesis of Modal Interfaces from higher-level specifications has been studied for the case of scenarios. In [242], Existential Live Sequence Charts are translated into Modal Specifications, hence providing a mean to specify modal contracts. It was recently shown in [192, 191] that the proposed model of Modal Interface Automata (MIA), a rich subset of Input-Output Modal Transition Systems (IOMTS) [176] featuring explicit output-must-transitions while input-transitions are always allowed implicitly, indeed possesses a conjunction. MIA are not restricted to be deterministic and revisit the model of IOMTS.

Regarding extensions, Acceptance Interfaces were proposed by J-B. Raclet [224, 225]. Informally, an Acceptance Interface consists of a set of states, with, for each state, a set of ready sets, where a ready set is a set of possible outgoing transitions from that state. Hence, each state of Acceptance Interfaces is labeled with a set of sets of transitions which explicitly specifies its set of possible models. Acceptance Interfaces are more expressive than Modal Interfaces but at the price of a prohibitive complexity for the various relations and operators of the theory. Modal Interfaces have been enhanced with marked states by Caillaud and Raclet [37]. Having marked states significantly improves expressiveness. It is possible to specify that some state must be reachable in any implementation while leaving the particular path for reaching it unspecified. As an example of use, Modal Interfaces with marked states have been applied in [43] to the separate compilation of multiple clocked synchronous programs.

Regarding extensions dealing with time, Timed Automata [9] constitute the basic model for systems dealing with time and built on top of automata. In words, timed automata are automata enhanced with clocks. Predicates on clocks guard both the states (also called “locations”) and the transitions. Actions are attached to transitions that result in the resetting of some of the clocks. Event-Clock Automata [10, 11, 49] form a subclass of timed automata where clock resets are not arbitrary: each action $\alpha$ comes with a clock $h_\alpha$ which is reset exactly when action $\alpha$ occurs. The interest of this subclass is that event-clock automata are determinizable, which facilitates the development of a (modal) theory of contracts on top of event-clock automata, seen as
corresponding components. A first interface theory able to capture the timing aspects of components is *Timed Interfaces* [107]. Timed Interfaces allows specifying both the timing of the inputs a component expects from its environment and the timing of the outputs it can produce. Compatibility of two timed interfaces is then defined and refers to the existence of an environment such that timing expectations can be met. The *Timed Interface* theory proposed in [102] fills a gap in the work introduced in [107] by defining a refinement operation. In particular, it is shown that compatibility is preserved by refinement. This theory also proposes a conjunction and a quotient operation and is implemented in the tool *Eđar* [101]. Timed Specification Theories are revisited from a linear-time perspective in [88]. The first *timed* extension of modal transition systems was published in [78]. It is essentially a timed (and modal) version of the Calculus of Communicating Systems (by Milner). Based on regions tool support for refinement checking were implemented and made available in the tool *Epsilon* [135]. Another timed extension of Modal Specifications was proposed in [50]. In this formalism, transitions are equipped with a modality and a guard on the component clocks, very much like in timed automata. For the subclass of modal event-clock automata, an entire algebra with refinement, conjunction, product, and quotient has been developed in [48, 49]. [174] addresses the problem of robust implementations in timed specification theories. The issue of time is also addressed in Metronomy [140] using a function-architecture separation, where timing contracts express constraints using tag-based expressions.

Resources other than time were also considered—with energy as the main target. *Resource Interfaces* [83] can be used to enrich a variety of interface formalisms (Interface Automata [105], Assume/Guarantee Interfaces [106], etc.) with a resource consumption aspect. Based on a two player game-theoretic presentation of interfaces, Resource Interfaces allow for the quantitative specification of resource consumption. With this formalism, it is possible to decide whether compositions of interfaces exceed a given resource usage threshold, while providing a service expressed either with Büchi conditions or thanks to quantitative rewards. Because resource usage and rewards are explicit rather than being defined implicitly as solutions of numerical constraints, this formalism does not allow one to reason about the variability of resource consumption across a set of logically correct models. Weighted
modal transition systems are proposed in [166, 30], in which each transition is decorated with a weight interval that indicates the range of concrete weight values available to the potential implementations. In this way resource constraints can be modeled using the modal approach. In the same direction, [31] proposes a novel formalism of label-structured modal transition systems that combines the classical may/must modalities on transitions with structured labels that represent quantitative aspects of the model. Last, the issue of contracts for heterogeneous systems is addressed in [186, 183, 184] by building on top of the tag machine component model [39].

Interfaces theories encompassing probability have been more recently proposed. Like the Interval Markov Chain (IMC) formalism [164] they generalize, Constraint Markov Chains (CMC) [71] are abstractions of a (possibly infinite) sets of Discrete Time Markov Chains. Instead of assigning a fixed probability to each transition, transition probabilities are kept symbolic and defined as solutions of a set of first order formulas. Variability across implementations is made possible not only with symbolic transition probabilities, but also thanks to the labeling of each state by a set of valuations or sets of atomic propositions. This allows CMCs to be composed thanks to a conjunction and a product operators. While the existence of a residuation operator remains an open problem, CMCs form an interface theory in which satisfaction and refinement are decidable, and compositions can be computed using quantifier elimination algorithms. In particular, CMCs with polynomial constraints form the least class of CMCs closed under all composition operators. In [113], the complexity of several problems for IMCs is studied. The complexity gap for thorough refinement of two IMCs and for deciding the existence of a common implementation for an unbounded number of IMCs is closed by showing that these problems are EXPTIME-complete. Abstract Probabilistic Automata (APA) [112] is another specification algebra with satisfaction and refinement relations, product and conjunction composition operators. Despite the fact that APAs generalize CMCs by introducing a labeled modal transition relation, deterministic APAs and CMCs coincide, under the mild assumption that states are labeled by a single valuation.
8.8. Bibliographical note

8.8.3 Features of our presentation

The presentation of interface theories in this monograph is new in many aspects. For the first time, all interface theories are clearly cast in the abstract framework of contracts following our meta-theory. In particular, the association, to an interface $\mathcal{C}$, of the two sets $\mathcal{E}_\mathcal{C}$ and $\mathcal{M}_\mathcal{C}$ is new. It clarifies a number of concepts. In particular, the interface theories inherit from the properties of the meta-theory without the need for specific proofs. The restriction operator for Modal Interfaces is new and so is its use in decomposing a contract into an architecture of subcontracts. The encoding of Assume/Guarantee reasoning in the framework of Modal Interfaces is also new. Note that this deeply relies on the meta-theory for its justification—inasmuch as the proposed formula provides a valid coding only under specific conditions regarding the tuple $(A_1, \ldots, A_n; G)$. Casting interface theories into the meta-theory was developed for the basic interface theories only. It would be useful to extend this to the different variants and see what the benefit would be. Benoît Caillaud has developed the MICA tool [70], which implements Modal Interfaces with all the operations and services discussed in this section.
9

Scheduling Contracts

9.1 Introduction

The focus of the contract framework developed in this chapter is the integration phase of a design process, where software components are allocated to a hardware platform. Typically software components share the computation and communication resources of the hardware platform. Access to these resources is arbitrated for example by an operating system, which applies a scheduling strategy to switch between different tasks to use a processor of the platform. This scheduling of software components obviously has a large impact on their timing behavior.

Often platform resources are considered quite late in a design process, when integration issues due to resource sharing can cause costly design iterations. The framework of scheduling contracts is geared towards supporting incremental design and independent implementability while taking resource usage of software components into account. A first version of this chapter was reported in [230].

9.1.1 An illustrative design scenario

We consider scenarios like the following: The bottom part of Figure 9.1 shows a target platform that is envisioned by, say, an Original Equipment
9.1. Introduction

Manufacturer (OEM). It consists of two processing nodes (CPU\textsubscript{1} and CPU\textsubscript{2}). Suppose the OEM wants to implement two applications, characterized by contracts \( C_1 \) and \( C_2 \), on this architecture and delegates their actual implementation to two different suppliers. Both applications share a subset of the resources of the target platform, e.g. tasks \( \tau_2 \) and \( \tau_4 \) are executed on CPU\textsubscript{2} after integration. Furthermore, we assume the system specification \( C \) shown in Figure 9.1 to be available from previous design phases. While some components together with their (local) contracts may also be known (e.g. in case of reuse), the OEM generally has to negotiate proper specifications with the suppliers, in our case the two contracts \( C_1 \) and \( C_2 \). At this point, the designer is faced with the following two issues:

1. The functions performed by the two subsystems must integrate correctly and their integration must satisfy the top-level function specification;

2. The scheduling of the software components delegated to each supplier must yield a satisfactory scheduling at system integration, meaning that timing constraints are met given the performance characteristics of the computing and communication resources, despite the two designs compete for shared resources.

For these two aspects, the design method must support independent development by each supplier while guaranteeing safe and correct integration, provided that the subsystems are correctly implemented. In this case study we
concentrate on issue 2, leaving aside issue 1 (the latter will be addressed by the parking garage case study of Chapter 10).

**Example 9.1.** Suppose task $\tau_1$ on the system depicted at the bottom of Figure 9.1 is a periodic task with period $p = 5$ and execution time $c = 3$. The two tasks $\tau_2$ and $\tau_3$ also have period $p = 5$. Task $\tau_2$ depends on $\tau_1$, i.e. is activated by $\tau_1$, and has an execution time $c_2 = 2$. Task $\tau_3$ depends on $\tau_2$ and has an execution time $c_3 = 1$. Task $\tau_4$ is also a periodic task with period $p_4 = 5$ and $c_4 = 2$. Suppose both CPUs are scheduled using a fixed priority preemptive policy, where tasks $\tau_1$ and $\tau_4$ have high priority on their CPU. The delay of the task-chain $\tau_1 \rightarrow \tau_2 \rightarrow \tau_3$ depends on the activation-pattern of $\tau_4$ and its execution time. This is illustrated in Figure 9.2. Once $\tau_1$ completes its execution, it activates task $\tau_2$ (via port $p_{o_1}$), which in turn might be preempted by $\tau_4$. Finally, $\tau_3$, activated by $\tau_2$, could be preempted by a subsequent instance of $\tau_1$ resulting from another event $i_1$ of the periodic event stream. An excerpt of a possible trace in $L$ is shown in Figure 9.2, which corresponds to the discussed scheduling scenario. Indeed, while composing the two subsystems and deploying them on the computing architecture of Figure 9.1, bottom, the main problem is the coupling due to the sharing of CPU2. Observe that every port and resource has its own event history in the component. Note that we omitted input ports connected to some output port: the intuition is that the
9.1. Introduction

synchronization of the subsystems’ behaviors is by unifying ports with identical names. This intuition for the composition of subsystems will be indeed formalized by our forthcoming notion of abstract scheduling component, see Definition 9.4.

9.1.2 Approach

In a design process involving an OEM/supplier chain, the OEM specifies the complete system at its architecture level. Details of the implementation are delegated to the different suppliers. Let us instantiate this on the task of generating a correct scheduling of the software components that we consider here. With reference to the design scenario of Section 9.1.1 and its Figure 9.1, the system architecture consists of the target platform and its resources. The following steps might be adopted in the design process:

1. Knowing the system architecture, the OEM specifies scheduling constraints and characteristics by only referring to the interfaces of the sub-systems. Detailed characteristics of the individual tasks are not known to the OEM.

2. On the basis of the sub-system specification provided by the OEM and the knowledge of the detailed characteristics of its local tasks, each supplier designs a local scheduling for the software components of its own sub-system.

3. It is then hoped that system integration will perform well.

Achieving this is likely to be difficult because real-time, which is at the heart of the problem, is a shared resource. Thus, any local design decision performed by a supplier may possibly affect the other sub-systems.

A conservative process, instead, consists in having the OEM perform system-level scheduling of all software components. A detailed specification of the restriction of this scheduling to each sub-system is then forwarded to the supplier in charge of it. The latter develops its sub-system accordingly. This, however, requires a rather fine grain knowledge by the OEM of the component characteristics, something not aligned with step 1 of the design process.
To improve this, compositional scheduling can be considered instead of global scheduling see [16]. The approach generally followed in these works is the following. The schedulability problem is structured as a hierarchy of subproblems. The solution of each subproblem is summarized using some form of interface (depending on the particular approach) and, at the next upper level, the amount of additional resource and deadline conditions is computed as the solution of some optimization problem. By relying on optimization, not games, such approaches relate to compositional verification, not contracts. They do not allow for an independent development by each supplier.

The community of Real-Time Calculus (RTC) proposed elegant interface models for real-time systems [247]. These frameworks, however, only apply to restricted classes of scheduling problems. This leads to inevitable over-approximations, except if the model of computation inherent to RTC is an adequate abstraction of the system under analysis. For example in RTC input or state dependent resource demands or schedules have to be approximated to express them in terms of interval-based arrival and service curves [172, 218, 171].

We believe it is preferable to have a flexible framework offering significant freedom for specifying the scheduling problem, including possible connections of it with other aspects of the system, e.g., the functions. The price for this is, of course, an added complexity in reasoning about such specifications. But we argue that the compositionality of contracts allows reducing that complexity. Our approach for this case study is therefore the following:

We assume that procedures performing global scheduling analysis are handy. Such procedures exist for various classes of scheduling problems. We will lift such procedures to a full fledged contract framework.

Our contract framework will be a mild adaptation of the Assume/Guarantee contracts (A/G-contracts for short) developed in Chapter 5.

Adapting Assume/Guarantee contracts: Recall that Assume/Guarantee contracts (A/G-contracts) are pairs of assumption and guarantees: \( \mathcal{C} = (A, G) \). In the basic A/G-contract framework [40], \( A \) and \( G \) are assertions, i.e., sets of traces for system variables.\(^1\) Components capturing legal implementations or

\(^1\)These are typically specified using modeling tools such as Simulink/Stateflow.
9.1. Introduction

Environments of contracts are also modeled by assertions. Component $E$ is a legal environment for $C$ if $E \subseteq A$ and component $M$ is an implementation for $C$ if $A \times M \subseteq G$. In this writing, $\subseteq$ is simply set inclusion and component composition $\times$ is by intersection of sets of traces (assuming that the underlying set of system variables is universal and thus fixed): $A \times M \overset{\text{def}}{=} A \cap M$.

One difficulty of $A/G$-contracts is the important notion of saturation: contracts $(A, G)$ and $(A, G \cup \neg A)$, where $\neg$ is set complement, possess identical sets of legal environments and implementations, so we consider them equivalent. The second one is called saturated and is a canonical form for the class of equivalent contracts. Also, $M_C = G \cup \neg A$ is the maximal implementation for this contract. Thus, we need the operations $\cup$ and $\neg$, or at least we need the operation $(A, G) \to G \cup \neg A$, which is to be interpreted as “$A$ entails $G$”. The Moore interfaces presented in Chapter 6 provide an effective way of implementing this entailment operation.

Component model: As a prior step, we need the right notion of component with its parallel composition. Ingredients of scheduling problems are: tasks with their precedence conditions reflecting data dependencies and resource allocation. The sets of timed traces we are interested in are those satisfying the scheduling constraints, plus extra quantitative properties such as period, deadline conditions, etc. Call concrete scheduling components the resulting model. Unfortunately, due to conflicts arising from shared resources, no rich algebra with the requested operators $\subseteq, \times, \cap, \cup, \neg$ exists in this model.

By abstracting away part of the description of task activities in traces, we slightly abstract concrete scheduling components to so-called abstract scheduling components. The idea is that we keep only what is essential for capturing interactions of scheduling problems, namely: 1) trigger and release events for tasks, and 2) busyness of resources. The abstraction map binds each concrete scheduling component to its abstraction and we will show that this binding is faithful. The framework of abstract scheduling components is simple enough so we manage to equip it with the wanted operations $\subseteq, \times, \cap, \cup, \neg$. $A/G$-contracts for abstract scheduling components follow then easily. In the next section we introduce our model of (concrete and abstract) scheduling components, and then, we develop scheduling contracts.
9.2 Scheduling components

This section is devoted to the introduction of concrete and abstract scheduling components. Then, we study the relation between them.

9.2.1 Concrete scheduling components

For our model of scheduling components we assume the following:

**A slotted model of real-time**, in which the real line \( \mathbb{R}_+ \) is divided into successive discrete time slots of equal duration. Successive slots are thus indexed by using natural numbers \( 1, 2, 3, \ldots, n, \ldots \in \mathbb{N} \), with 0 indexing the initial conditions. In the sequel, the term “date” will refer to the index of the time slot in consideration.

**An underlying set \( T \) of tasks**, generically denoted by the symbol \( \tau \). Each task \( \tau \) comes equipped with a pair \((p^t(\tau), p^c(\tau))\) \( \in \mathcal{P} \times \mathcal{P} \) of trigger and completion ports, where \( \mathcal{P} \) is an underlying set of ports. For \( \tau_1, \tau_2 \in T \), say that \( \tau_1 \) *precedes* task \( \tau_2 \), written

\[
\tau_1 \rightarrow \tau_2,
\]

if the completion port of \( \tau_1 \) coincides with the start port of \( \tau_2 \): \( p^c(\tau_1) = p^t(\tau_2) \).

The dual relation between ports will also be needed: for \( p_1, p_2 \in \mathcal{P} \), say that \( p_1 \) *precedes* \( p_2 \), written

\[
p_1 \rightarrow p_2,
\]

if there exists a task \( \tau \in T \) such that \( p_1 = p^t(\tau) \) and \( p^c(\tau) = p_2 \). Finally, for \( p \in \mathcal{P} \), we define its *preset* \( \preceq_p \) and *postset* \( \succeq_p \) as being the sets of tasks \( \tau \) such that \( p^t(\tau) = p \) and \( p^c(\tau) = p \), respectively.

**The following alphabets describing task events:**

- The *control alphabet* \( \Sigma_c = \{ t, c, tc, aw, sl \} \) collects the *trigger*, *completion*, *trigger-and-completion*, *awake*, and *sleeping* events, for a task; this alphabet describes the triggering and completion of tasks; since
9.2. Scheduling components

We follow a slotted model of time, triggering and completion can occur within the same slot, which is indicated by the event $tc$;\(^2\)

- The busyness alphabet $\Sigma_b = \{\beta, \iota\}$ collecting the busy and idle events; this alphabet indicates, for a task, its status busy/idle at a given time slot.

On top of these alphabets, we build:

$$\Sigma = \text{def}\{ (c, b) \in \Sigma_c \times \Sigma_b \mid c = \text{sl} \Rightarrow b = \iota \}$$

(9.3)

reflecting that task $\tau$ can only be busy when it is not sleeping. The status of each task in each time slot is expressed by using alphabet $\Sigma$. This is illustrated on Figure 9.3.

An underlying set $\mathcal{R}$ of resources, generically denoted by the symbol $r$. A resource can be either available or busy with executing a given task at a given time slot. Resources can run in parallel. Each resource $r \in \mathcal{R}$ is assigned the alphabet $\Sigma_r \subseteq \mathcal{T} \cup \{0\}$ of the tasks it can run, where the special symbol 0 indicates that $r$ is idle.

We are now ready to introduce concrete scheduling components:

**Definition 9.1** (concrete scheduling component). A concrete scheduling component is a pair $M = (K, L)$, where:

- $K = (T, R, \rho)$ is the sort of $M$, where: $T \subseteq \mathcal{T}$ is the set of tasks, $R \subseteq \mathcal{R}$ is the set of resources, and $\rho : T \rightarrow R$, the resource allocation map, is a partial function satisfying $\tau \in \Sigma_{\rho(\tau)}$. Say that tasks $\tau_1$ and $\tau_2$ are

\(^2\)Strictly speaking, statuses $aw$ and $sl$ add no useful information about the history of a task; these two statuses are only here for technical convenience. They are used in (9.3) to build our structured alphabets and they facilitate the formulation of the condition 9.6 characterizing behaviors in Definition 9.2.
non-conflicting if they do not use the same resource:

\[
\tau_1 \parallel_K \tau_2 \quad \text{if} \quad \begin{cases} \\
either \rho(\tau_1) \text{ is undefined} \\
or \rho(\tau_2) \text{ is undefined} \\
or \rho(\tau_1) \neq \rho(\tau_2) \end{cases}
\] (9.4)

We require that the restriction, to \(T\), of the relation \(\rightarrow\) introduced in (9.1) is circuit free and we denote by \(\preceq\) the induced partial order, which we call the precedence order on \(T\). Relation \(\rightarrow\) introduced in (9.2) is circuit free if so was \(\rightarrow\) and, with no risk of confusion, we also denote by \(\preceq\) the precedence order on \(P_{T}\) generated by \(\rightarrow\).

- \(L \subseteq (T \rightarrow \Sigma)^\omega\) is the language of \(M\), where \(A^\omega\) denotes the set of all infinite words over alphabet \(A\). Due to the decomposition (9.3) of \(\Sigma\), every word \(w \in L\) can be equivalently seen as a tuple of pairs of words \(w = (w_c(\tau), w_b(\tau))_{\tau \in T}\) describing the control and busyness history of task \(\tau\) in word \(w\). For \(T' \subseteq T\) we define \(w(T') = (w_c(\tau), w_b(\tau))_{\tau \in T'}\).

A possible word \(w\) of \(L\) was shown in Figure 9.2, together with its corresponding (non-conflicting) resource usage.

Since we do not restrict the class of languages \(L\) we consider, our component model is fully general, as announced.

Notations: For \(v \in \{0, 1\}^\omega\) a word representing a flow of events (1 represents the occurrence of the event), we denote by \(\{N_n(v) \mid n \geq 0\}\) the successive instants of occurrence of the event in \(v\):

\[N_0(v) = 0 \text{ and, for } n > 0, N_n(v) = \min\{m > N_{n-1}(v) \mid v(m) = 1\}.
\]

We then denote by

\[N_n^w(w, \tau) \quad \text{and} \quad N_n^c(w, \tau)\] (9.5)

the successive instants of occurrence of the events “\(w_c(\tau) = t\) or \(tc\)” and “\(w_c(\tau) = tc\) or \(c\)”, respectively.

Not all words of \(L\) are compliant with the rules of scheduling. We characterize those compliant words in the following definition, where \(M = (K, L)\) denotes a concrete scheduling component:
Definition 9.2 (semantics of a concrete scheduling component). A behavior of sort $K$ is any infinite word $w \in (T \rightarrow \Sigma)^\omega$ satisfying the following three scheduling conditions:

\[
\forall \tau \in T : w_c(\tau) \in L =_{\text{def}} (sl^\tau.(tc + t.aw^*c))^\omega, \quad (9.6)
\]
\[
\forall \tau, \tau' \text{ s.t. } \tau \leq \tau' \implies \forall n > 0 : N^\tau_n(w, \tau) < N^\tau'_n(w, \tau') \quad (9.7)
\]
\[
\forall \tau, \tau' \text{ s.t. } \neg[\tau \parallel K \tau'] \implies \neg[\exists n : w_b(n, \tau) = w_b(n, \tau') = \beta] \quad (9.8)
\]

The semantics of $M$ is the sublanguage $[M] \subseteq L$ consisting of all behaviors of $K$ belonging to $L$. Say that $M$ is schedulable if $[M] \neq \emptyset$. □

In words, the above three conditions mean:

(9.6) The two events $t$ and $c$ alternate in $w_c(\tau)$, with $t$ occurring first; $tc$ is interpreted as the immediate succession of two $t$ and $c$ events at the same time slot. Recall that $a^* =_{\text{def}} \epsilon + a + a.a + a.a.a + \ldots$ is the Kleene closure starting at the empty word.

(9.7) $\tau \leq \tau'$ implies that $\tau'$ can only start after $\tau$ has been completed; note that it is enough to check (9.7) for pairs $\tau, \tau'$ such that $\tau \rightarrow\rightarrow \tau'$;

(9.8) $w$ is non-conflicting, meaning that, for any two conflicting tasks $\tau$ and $\tau'$ belonging to $T$ (cf. (9.4)), it never happens that $w(\tau)$ and $w(\tau')$ are busy at the same time slot.

Observe that, due to the above Condition (9.7), tasks related by precedence conditions possess identical logical clocks. This is not required for tasks not related by precedence conditions.

Comment 9.1 (on the set $T$ of underlying tasks). Tasks are considered black-box. The underlying set of tasks is assumed, with their associated dependencies. It corresponds to the actual tasks that a designer could specify by using some language aimed at describing the system functions. We are not interested in this functional aspect of system design in this chapter. □

Comment 9.2 (roles of $K$ and $L$). The pair $M = (K, L)$ can be seen as the specification of a global scheduling problem. Sort $K$ fixes the set of tasks, the set of resources, and the allocation of tasks to resources. The language $L$ can serve to specify additional aspects of this scheduling problem, including task
durations and/or minimum time interval between successive activation calls for a task, see Section 9.5 on modeling methodology.

**Comment 9.3 (role of $[[M]]$).** Semantics $[[M]]$ can be seen as the maximally permissive solution of the scheduling problem stated by $M$. Recall that, throughout this use case, we assume that the procedure for computing the semantics of a scheduling component is available. Our aim is to lift such procedures to a contract framework supporting compositionality and independent development.

The class of concrete scheduling components is easily equipped with a parallel composition:

**Definition 9.3 (composition of concrete scheduling components).** Say that $M_1$ and $M_2$ are *composable* if their allocation maps $\rho_1$ and $\rho_2$ coincide on $T_1 \cap T_2$ and the relation $\rightarrow_1 \cup \rightarrow_2$ on $T_1 \cup T_2$ is cycle free. If $M_1$ and $M_2$ are composable, their *composition* is defined by $M_1 \times M_2 = \text{def} (K_1 \times K_2, L_1 \times L_2)$, where $K_1 \times K_2 = (T, R, \rho)$ is defined as follows:

- $T = T_1 \cup T_2$
- $R = R_1 \cup R_2$
- $\forall \tau \in T : \rho(\tau) = \text{if } \tau \in T_1 \text{ then } \rho_1(\tau) \text{ else } \rho_2(\tau)$

and

$$L_1 \times L_2 = \text{pr}_{T \rightarrow T_i}^{-1}(L_1) \cap \text{pr}_{T \rightarrow T_i}^{-1}(L_2)$$

where $\text{pr}_{T \rightarrow T_i}()$, $i = 1, 2$, denotes the projection from $T$ to $T_i$ and $\text{pr}^{-1}$ is its inverse.

Of course, the key to understand the meaning of composition $\times$ is the construction of the semantics $[[M_1 \times M_2]]$, where the scheduling problem is solved. To each scheduling component $M = (K, L)$, we associate the following scheduling component where $L$ is replaced by the semantics $[M]$ of $M$:

$$[M] = \text{def} (K, [M]) \quad (9.9)$$

The following result holds:

**Lemma 9.4.** For $M_1$ and $M_2$ composable, $[[M_1 \times M_2]] = [[M_1]] \times [[M_2]]$. 


9.2. Scheduling components

Proof. Since sorts are unchanged, from \( M_i \) to \([M_i]\), the right hand side is well defined. By definition of the parallel composition \( \times \) and using the scheduling conditions of Definition 9.2, \([M_i \times M_j]\) is the set of words \( w \) such that:

\[
\exists w_i \in L_i, i = 1, 2 \quad \text{s.t.} \quad \forall \tau \in T_i \Rightarrow w(\tau) = w_i(\tau)
\]

\[
\forall \tau \in T : w_c(\tau) \in \mathcal{L}
\]

\[
\forall \tau, \tau' \text{ s.t. } \tau \leq \tau' \quad \Rightarrow \quad \forall n > 0 : N_n^c(w, \tau) < N_n^c(w, \tau')
\]  

\[
\forall \tau, \tau' \text{ s.t. } \neg[\tau \parallel K \tau'] \quad \Rightarrow \quad \neg[\exists n : w_b(n, \tau) = w_b(n, \tau') = \beta]
\]  

(9.10)

On the other hand, \([M_i \times M_j]\) is the set of words \( w \) satisfying the following conditions, where \( i, j \in \{1, 2\} \) and \( \tau_i, \tau_j \) denote generic tasks in \( T_i \):

\[
\exists w_i \in L_i \quad \text{s.t.} \quad \forall \tau \in T_i \Rightarrow w(\tau) = w_i(\tau)
\]

\[
\forall \tau \in T_i : w_i(\tau) \in \mathcal{L}
\]

\[
\forall \tau_i, \tau_j' \text{ s.t. } \tau_i \leq \tau_j' \quad \Rightarrow \quad \forall n > 0 : N_n^c(w_i, \tau_i) < N_n^c(w_i, \tau_j')
\]  

\[
\forall \tau_i, \tau_j' \text{ s.t. } \neg[\tau_i \parallel K \tau_j'] \quad \Rightarrow \quad \neg[\exists n : w_{i,b}(n, \tau_i) = w_{i,b}(n, \tau_j') = \beta]
\]

\[
\forall \tau_i, \tau_j' \text{ s.t. } \neg[\tau_i \parallel K \tau_j'] \quad \Rightarrow \quad \neg[\exists n : w_{j,b}(n, \tau_i) = w_{j,b}(n, \tau_j') = \beta]
\]  

(9.11)

The first two conditions of (9.10) and (9.11) coincide, and so do the third conditions since it is enough to check it on the relation \( \rightarrow \rightarrow = \rightarrow \rightarrow_1 \cup \rightarrow \rightarrow_2 \).

Finally, the fourth condition of (9.10) decomposes as the fourth and fifth conditions of (9.11). The fifth condition of (9.11) is enforced when mapping the composition \([M_i \times M_j]\) to its semantics \([M_i \times M_j]\). This proves the lemma. □

Comment 9.5 (the need for a more abstract framework). As announced in the introductory discussion of Section 9.2, the model of concrete scheduling component is too complex and detailed as a model of component on top of which contracts can be built. In particular, the consideration of sorts raises a difficult typing problem and is an obstruction against getting the constructs required for a universe of components. Moreover, scheduling component \( M \) and its semantics \([M]\) are related in a complex way, through Definition 9.2. This makes it difficult to define the operations we need on components, particularly \( \subseteq \) and \( \cup\neg \) (in turn, parallel composition \( \times \) was easy to define as we have seen). The notion of abstract scheduling component we develop in the forthcoming section will overcome these difficulties. Abstract scheduling
components capture the architecture aspect of Figure 9.1, namely: ports carrying start and completion events of tasks, and resources — tasks themselves are, however, abstracted away.

\[ \square \]

### 9.2.2 Abstract Scheduling Components

In the following, we set \( \mathbb{B} = \{0, 1\} \), and \( \Sigma_R = \prod_{r \in R} \Sigma_r \). Recall that \( \Sigma_r \) is the alphabet of tasks that can be executed by resource \( r \), see the beginning of Section 9.2.1. We will freely interpret \( \{0, 1\} \) as the Boolean domain of events and symbol “1” indicates the occurrence of an event.

**Definition 9.4 (abstract scheduling component).** An abstract scheduling component is a language \( M \subseteq \mathcal{V}^w \), where \( \mathcal{V} = \mathbb{B}^P \times \Sigma_R \). Abstract scheduling components are equipped with the following algebra:

- Boolean algebra \( \cap, \cup, \neg \) (set complement), and set inclusion \( \subseteq \);

- A parallel composition by intersection: \( M_1 \times M_2 = \mathbb{B}^P \times \Sigma_R \). \( \square \)

Thus, abstract scheduling components offer all the algebra required for a universe of components on top of which A/G-contracts can be built. It is therefore interesting to map concrete to abstract scheduling components. Note that the operation of contract saturation can be implemented using the framework of Moore Interfaces developed in Chapter 6.

### 9.2.3 Mapping Concrete to Abstract Scheduling Components

Recall that, for \( K = (T, R, \rho) \) a sort, we denote by \( P = \mathbb{B}^P \times \Sigma_R \) the set of ports used by \( T \), see the beginning of Section 9.2.1. Then, we set

\[ \mathcal{V}_K = \mathbb{B}^P \times \Sigma_R \], where \( \Sigma_R = \prod_{r \in R} \Sigma_r \quad (9.12) \]

**Definition 9.5 (mapping concrete to abstract scheduling components).** Each concrete scheduling component \( M = (K, L) \) is mapped to a unique abstract scheduling component \( \llbracket M \rrbracket^A \) called its abstract semantics, defined as follows, where \( w \) denotes an arbitrary behavior of \( M \), i.e., and element of
9.2. Scheduling components

\[[M]\]:

\[
V_P(w) = \left\{ v_P \in (\mathbb{B}^P)^\omega \mid \forall n \in \mathbb{N}, \forall p \in P, \forall (\tau, \tau') \in p \times p^* \downarrow N_n^p(w, \tau) \leq N_n^p(v_P, p) < N_n^p(w, \tau') \right\}
\]

(9.13)

\[
V_R(w) = \left\{ v_R \in (\Sigma_R)^\omega \mid \forall n \in \mathbb{N}, \downarrow \text{if } \exists \tau \in T : w_p(n, \tau) = \beta \text{ then } v(n, p(\tau)) = \tau \text{ else } \left[ \exists \tau' \in T \setminus T \text{ : } v(n, r') = \tau' \right] \right\}
\]

(9.14)

and finally,

\[
\llbracket M \rrbracket^A = \text{def } \left\{ v = (v_P, v_R) \in V^{\omega} \mid \exists w \in \llbracket M \rrbracket \text{ s.t.: } v_P \in V_P(w) \quad v_R \in V_R(w) \right\}
\]

(9.15)

defines the abstract semantics of \(M\). □

Observe that set \(V_P(w)\) is non empty if and only if \(w\) satisfies the second and third conditions of (9.10), and set \(V_R(w)\) is non empty if and only if \(w\) satisfies the fourth condition of (9.10). In words, the construction (9.15) proceeds by the following steps:

1. Pick any \(w \in \llbracket M \rrbracket\), see Definition 9.2;
2. Denote by \(\pi_T(w)\) the word over \(\{0, 1\}^{P_T}\) obtained from \(w\) as follows:

\[
\forall p \in P_T, \text{ define } \ast p = \{ \tau \in T \mid p^*(\tau) = p \} \quad \ast p^* = \{ \tau \in T \mid p(\tau) = p \},
\]

the sets of anterior and posterior tasks of \(p\). Put the \(n\)th event of \(p\), nondeterministically:

- after the \(n-1\)st event of \(p\),
- when or after every task belonging to \(\ast p\) has completed for the \(n\)th time in \(w\), and
- strictly before every task belonging to \(p^*\) has started for the \(n\)th time in \(w\).

The first condition is not considered if \(\ast p = \emptyset\) and the second condition is not considered if \(p^* = \emptyset\).
3. Denote by $\pi_R(w)$ the word over alphabet $\prod_{r \in R} \Sigma_r$ defined as follows:

(a) For every time slot $n$ and every resource $r \in R$, set

$$\pi_R(w)(r, n) = \tau$$

if and only if $w(\tau, n) = (c, b)$ satisfies $b = \beta$ and $\rho(\tau) = r$.

This part of word $\pi_R(w)$ represents the “positive history” of $w$, that is, the use of the resources belonging to $R$ by tasks belonging to $T$; it is reflected by the “then” alternative in (9.14);

(b) We complement $\pi_R(w)$ by describing the “negative history” of $w$, consisting of a description of all the possibilities left, for tasks not belonging to $T$, in using resources from $R$:

in all slots of $\pi_R(w)(r, n)$ that are kept idle after step 3a, we set $\pi_R(w)(r, n) = \tau'$ where $\tau' \in \Sigma_r, \tau' \not\in T$ is chosen nondeterministically.

The negative history is reflected by the “else” alternative in (9.14).

Then, with reference to the sort $K = (T, R, \rho)$ of $M$, we set:

$$\eta_K(w) = \text{def} \ (\pi_T(w), \pi_R(w)) \in \mathcal{V}_K^{\omega}$$

(9.16)

4. Finally, we define

$$\llbracket M \rrbracket^{A} = \text{def} \ \text{pr}_{\mathcal{V}_K^{\omega}}^{-1} \left( \left\{ \eta_K(w) \big| w \in \llbracket M \rrbracket \right\} \right) \subseteq \mathcal{V}_K^{\omega}$$

where the quantification ranges over $w \in \llbracket M \rrbracket$ and all instances of nondeterministic choices in steps 2 and 3, and $\text{pr}_{\mathcal{V}_K^{\omega}}$ denotes the projection, from $\mathcal{V}_K^{\omega}$ to $\mathcal{V}_K^{\omega}$.

Step 2 of this construction is sound since $w$ is a behavior in the sense of Definition 9.2. Steps 2 and 3 are the key steps.

Step 2 transforms a max-plus type of parallel composition (every task waits for all its preceding tasks having completed before starting) into a dataflow connection where data are communicated through the shared ports. The data communicated are the events carried by the ports. These events occur nondeterministically after all preceding tasks have completed for the $n$th time and before all succeeding tasks start for the $n$th time.
9.2. Scheduling components

Step 3 complements the actual history of each task of \( M \) by an explicit description of all possibilities that are left to other scheduling components in using resources shared with \( M \). The reason for doing this is that this allows to capture the interleaved use of shared resources by different components, by a simple parallel composition by intersection. The price for this is that one behavior of the concrete semantics of \( M \) is mapped to a set of behaviors of its abstract semantics.

\[
\begin{align*}
\tau_1, \tau_3 & \text{ are assigned to CPU}_1 \\
\tau_2 & \text{ is assigned to CPU}_2
\end{align*}
\]

Figure 9.4: Showing a concrete behavior of \( M \) (left, with reference to Figure 9.2) and a corresponding abstract behavior of \( \llbracket M \rrbracket^A \) (right), by using \( P = \{i_1, o_1, o_2, o_3\} \) as underlying alphabet of ports. On the second diagram, blanks figure the slots left free for any external task to run on the referred resource. The yellow rectangles indicate the room for nondeterministic choices; bounds of these rooms are figured by pointing arrows; where such arrow is missing, the corresponding rectangle is unbounded on that side.

The above construction is illustrated in Figure 9.4. When hiding the tasks sitting inside the boxes, the architecture shown on Figure 9.1 is a dataflow representation of \( \llbracket M \rrbracket^A \): in interpreting this figure, one should consider that each task is free to start any time after it has received its triggering event, and free to wait for some time before emitting its completion event.

Lemma 9.6. The mapping \( M \rightarrow \llbracket M \rrbracket^A \) satisfies the following properties:

1. Schedulability is preserved in that \( \llbracket M \rrbracket \neq \emptyset \) if and only if \( \llbracket M \rrbracket^A \neq \emptyset \);

2. For every \( r \not\in R \), the set \( \{v(r) \mid v \in \llbracket M \rrbracket^A\} \) is the free language \( (T - T)^\omega \).
Important notice: The special property 2 is not preserved under the Boolean set algebra. Therefore, the mapping $M \rightarrow \llbracket M \rrbracket^A$ is not surjective.

Faithfulness of the mapping

Consider two concrete scheduling components $M$ and $M'$. Checking the inclusion $\llbracket M' \rrbracket^A \subseteq \llbracket M \rrbracket^A$ requires computing the two abstract semantics, which may be costly. In this section we provide sufficient conditions for this inclusion, to be checked directly on the concrete scheduling components.

Lemma 9.7. The following conditions on the pair $(M', M)$ are sufficient to ensure $\llbracket M' \rrbracket^A \subseteq \llbracket M \rrbracket^A$:

1. There exists a surjective total map $\psi : T' \rightarrow T$, such that:

   (a) For every $\tau \in T$:
   
   $$ p^i(\tau) = \min_{\psi(\tau')=\tau} p^i(\tau') \quad \text{and} \quad p^e(\tau) = \max_{\psi(\tau')=\tau} p^e(\tau') \quad (9.17) $$

   where $\min$ and $\max$ refer to the order on the ports of $M'$;

   (b) The following holds, for every 4-tuple of tasks $(\tau'_1, \tau'_2, \tau_1, \tau_2) \in T'^2 \times T^2$:
   
   $$ \left[ \psi(\tau'_1) = \tau_1 \text{ and } \psi(\tau'_2) = \tau_2 \right] \implies \left[ \tau'_1 \parallel' \tau'_2 \Rightarrow \tau_1 \parallel \tau_2 \right] \quad (9.18) $$

2. The two languages $L'$ and $L$ are related by

   $$ L' \subseteq \left\{ w' \in (T' \rightarrow \Sigma)^\omega \mid \exists w \in L \text{ s.t. } w'(\tau') = w(\psi(\tau')) \right\} \quad (9.19) $$

Furthermore, the following condition implies (9.18):

$$ \rho'(\tau') = \rho(\psi(\tau')). \quad (9.20) $$

Say that $M' \sqsubseteq M$ when the conditions (9.17–9.19) hold. □

Observe that Conditions 1 involve only the sorts $K$ and $K'$ of $M$ and $M'$. Condition 9.19 formalizes the situation in which the language $L'$ is specified through timing properties relating certain events of interest for tasks of $M'$ (duration between trigger and completion, end-to-end duration when traversing a set of successive tasks, etc.). The considered events are then mapped to some events of $M$ and the timing property remains the same or is strengthened.
9.2. Scheduling components

Proof. The additional statement is obvious. Note that (9.20) implies $R' = R$. So, we focus on the main statement. By Condition 9.19, we only need to prove that the scheduling conditions associated to $K_1$ are stronger than those associated to $K_2$. By definition of the precedence order, see (9.1), Condition 1a implies that, for every 4-tuple of tasks $(\tau_1', \tau_2', \tau_1, \tau_2) \in T'^2 \times T^2$,

$$\left[ \psi(\tau_1') = \tau_1 \text{ and } \psi(\tau_2') = \tau_2 \right] \Rightarrow \left[ \tau_1 < \tau_2 \Rightarrow \tau_1' < \tau_2' \right]. \quad (9.21)$$

Set $\Psi = (\psi, \psi)$. By (9.17), $M$ involves a subset of the ports of $M'$. By (9.18), we have $\| \subseteq \Psi^{-1}(\|)$, and, by (9.21), we have $\prec' \supseteq \Psi^{-1}(\prec)$. Consequently, the ports involved in $M$ are less sequentialized and more concurrent than the same ports in $M'$. Furthermore, additional ports only involved in $M'$ may be subject to precedence constraints and access conflicts. The inclusion $\| M' \| \subseteq \| M \|$ follows.

The following result expresses that abstract semantics is faithful with respect to concrete scheduling components equipped with the composition $\times$ introduced in Definition 9.3:

Lemma 9.8. If concrete scheduling components $M_1$ and $M_2$ are composable, then $\| M_1 \times M_2 \| = \| M_1 \| \times \| M_2 \|$ holds.

Proof. To construct $\| M_1 \times M_2 \|$ the following steps are performed:

1. Pick all pairs $(w_1, w_2) \in L_1 \times L_2$;

2. Keep only the pairs $(w_1, w_2)$ that agree on $T_1 \cap T_2$; for such a pair $(w_1, w_2)$, fuse $w_1$ and $w_2$ by setting for $\tau \in T_i, i = 1, 2$: $w(\tau) := w_i(\tau)$;

3. Keep only the words $w$ that are behaviors of sort $K = (T, R, \rho)$, thus obtaining $\| M_1 \times M_2 \|$;

4. For each remaining word $w$, following steps 2 and 3 of Definition 9.5, generate non-deterministically $v_K = \text{def } \eta_K(w)$;

5. Expand $v_K$ to all ports and resources by applying the inverse projection of step 4 of Definition 9.5.

To construct $\| M_1 \| \times \| M_2 \|$ the following steps are performed:
Scheduling Contracts

1. Pick all pairs \((w_1, w_2) \in L_1 \times L_2\);

2. Keep only the pairs \((w_1, w_2)\) such that \(w_i\) is a behavior of sort \((T_i, R_i, \rho_i)\), for \(i = 1, 2\);

3. For each remaining word \(w_i\), following steps 2 and 3 of Definition 9.5, generate non-deterministically \(v_K = \text{def } \eta_K(w_i)\);

4. Keep only the pairs \((v_{K_1}, v_{K_2})\) that agree on \((P_{T_1} \cap P_{T_2}) \cup (R_1 \cap R_2)\); for such a pair, fuse \(v_{K_1}\) and \(v_{K_2}\) by setting, for \(p \in P_{T_i}, r \in R_j\) for \(i, j = 1, 2\): \(v_K(p) = v_{K_i}(p)\) and \(v_K(r) = v_{K_j}(r)\);

5. Expand \(v_K\) to all ports and resources by applying the inverse projection of step 4 of Definition 9.5.

The first and last steps of these two procedures are identical. On the other hand, we claim that the mapping \(M \mapsto [M]^{\mathbb{A}}\) specified in step 2 of Definition 9.5 is indeed designed so that chaining the second, third, and fourth steps of the above two procedures yields identical results. We show this next. To this end, we will need to access subformulas of some multiple formulas: these will be labeled with an extra label \(i, ii, \) etc.

The abstract semantics \([M_1 \times M_2]^{\mathbb{A}}\) is obtained by: 1) applying the formulas (9.10–i, ii, iii, iv) defining the concrete semantics of \(M_1 \times M_2\), followed by: 2) submitting the result to the formulas (9.13), (9.14) and (9.15) defining the mapping \(M \mapsto [M]^{\mathbb{A}}\).

On the other hand, the composition \([M_1]^{\mathbb{A}} \times [M_2]^{\mathbb{A}}\) is obtained by: 1) applying the formulas (9.10–ii, iii, iv) to get the concrete semantics of each \(M_i\) separately, 2) applying (9.13), (9.14) and (9.15) to each \([M_i]^{\mathbb{A}}\), and finally, applying the composition by intersection \(([M_1]^{\mathbb{A}} \times [M_2]^{\mathbb{A}}) \mapsto [M_1]^{\mathbb{A}} \times [M_2]^{\mathbb{A}}\).

Applying the formulas (9.10–ii, iii, iv) results in identical pre-selections of words \(w_i, i = 1, 2\) in both \((a)\) and \((b)\). The only properties that remain to be proved are the following two equivalences, regarding ports and resources:
9.2. Scheduling components

Properties regarding ports: \((9.22)\Leftrightarrow (9.23)\), where \(w\) is any behavior of \([[M_1 \times M_2]]\) and \(w(\tau) = w_i(\tau)\) for every \(\tau \in T_i\):

\[
\forall n \in \mathbb{N}, \forall p \in \mathcal{P}, \forall (\tau, \tau') \in (T \times T) \cap (\ast \times \ast) \\
\implies N^n_0(w, \tau) \leq N_n(v_p, p) < N^n_0(w, \tau') \quad (9.22)
\]

\[
\forall i=1, 2, \forall n \in \mathbb{N}, \forall p \in \mathcal{P}, \forall (\tau, \tau') \in (T_i \times T_i) \cap (\ast \times \ast) \\
\implies N^n_0(w_i, \tau) \leq N_n(v_p, p) < N^n_0(w_i, \tau') \quad (9.23)
\]

In (9.23), the superscript \(\ast\) refers to the dependence relation between ports and tasks in component \(M_i\), see (9.2). Since \(\rightarrow \rightarrow = \rightarrow_1 \cup \rightarrow_2\), condition \((9.22)\) implies \((9.23)\), and coincides with it when \(p \in P_i\) and \((\tau, \tau') \in (T_i \times T_i)\), i.e., when ports and tasks are local to the same component. It remains to prove that \((9.23)\) implies \((9.22)\) even for the case in which \((\tau, \tau') \in (T_j \times T_i)\), for \(j \neq i\). To derive this, we combine \((9.23)\) for the component \(M_j\) and port \(p = p^j(\tau)\) with \((9.23)\) for the component \(M_i\) and the same port \(p = p^j(\tau')\).

Properties regarding resources: \((9.24)\Leftrightarrow (9.25)\), where \(w\) is any behavior of \([[M_1 \times M_2]]\) and \(w(\tau) = w_i(\tau)\) for every \(\tau \in T_i\):

\[
\forall n \in \mathbb{N} \implies \begin{cases} 
\text{if } \exists \tau \in T : w_b(n, \tau) = \beta \\
\text{then } v(n, \rho(\tau)) = \tau \\
\text{else } [\exists \tau' \in \mathcal{T} \setminus T, \exists \tau' : \tau' \in \Sigma_{r'}] \implies v(n, \tau') = \tau'
\end{cases} \quad (9.24)
\]

\[
\forall i=1, 2, \forall n \in \mathbb{N} \implies \begin{cases} 
\text{if } \exists \tau \in T_i : w_{i,b}(n, \tau) = \beta \\
\text{then } v(n, \rho_i(\tau)) = \tau \\
\text{else } [\exists \tau' \in \mathcal{T} \setminus T_i, \exists \tau' : \tau' \in \Sigma_{r'}] \implies v(n, \tau') = \tau'
\end{cases} \quad (9.25)
\]

Since the parallel composition is by intersection, the desired equivalence follows. This finishes the proof of the lemma. \(\Box\)

For \((A, G)\) a composable pair of concrete scheduling components, we will sometimes need to consider the expression \([[G]]^A \cup \neg [[A]]^A\). This is addressed in the following lemma, where the items of \(A\) possess “\(A\)” as a subscript, and similarly for \(G\):

**Lemma 9.9.** Let \((A, G)\) be a composable pair of concrete scheduling components such that \(R_A = \emptyset\). Then, the following formulas define a concrete scheduling component \(M = ((T, R, \rho), L)\) such that \([[M]]^A = [[G]]^A \cup \neg [[A]]^A\):

\[
T = T_A \cup T_G, \quad R = R_G, \quad \rho(\tau) = \begin{cases} 
\text{if } \tau \in T_G \text{ then } \rho_G(\tau), \\
\text{else } \rho_A(\tau)
\end{cases} \quad (9.26)
\]

\[
L = \text{pr}^{-1}_{T \rightarrow T_G}(L_G) \cup \text{pr}^{-1}_{T \rightarrow T_A}(\neg L_A) \quad (9.27)
\]
where $\text{pr}_{T \rightarrow T}^{-1}()$ and $\text{pr}_{T \rightarrow T}^{-1}()$ denotes the referred inverse projections. □

Proof. This results from the fact that, thanks to the assumption that $R_A = \emptyset$, the characterization (9.6) for the abstract scheduling components that are image of a concrete scheduling component is satisfied. □

Recall, though, that an alternative approach to addressing contract saturation is by using the Moore game of Moore Interfaces, see Chapter 6.

9.3 Scheduling contracts

9.3.1 Abstract and concrete scheduling contracts

In this section we adapt Chapter 5. As recommended in that chapter, we first define what components are for this theory, and then we define contracts. Regarding components, the notations used here refer to the operations $\subseteq$, $\cap$, $\cup$, $\neg$, $\times$ introduced for abstract scheduling components in Section 9.2.2.

Definition 9.6 (components and contracts). A scheduling contract is a pair $C = (A, G)$ of abstract scheduling components, called its assumptions and guarantees, respectively.

The set $E_C$ of the legal environments for $C$ collects all abstract scheduling components $E$ with non-empty semantics and such that $E \subseteq A$. The set $M_C$ of all implementations of $C$ collects all abstract scheduling components $M$ with non-empty semantics and such that $A \times M \subseteq G$.

Each scheduling contract can be put in its equivalent saturated form $C = (A, G \cup \neg A)$, possessing the same sets of legal environments and implementations. Scheduling contract $C$ is compatible if and only if $A \neq \emptyset$ and consistent if and only if $G \cup \neg A \neq \emptyset$. Say that scheduling contract $C = (A, G)$ is schedulable if $A \cap G \neq \emptyset$. □

The justification of this notion of schedulability for contracts is given in the next section. The material of Section 5.1 applies verbatim and the reader is referred to it. Note that $A \cap G = A \cap (G \cup \neg A)$, hence checking schedulability does not require the contract to be saturated.

In practice the designer will specify scheduling contracts via pairs $(A, G)$ of composable concrete scheduling components, called its assumption and guarantee (see Definition 9.3 for the notion of composability). Thus, define:
### 9.3. Scheduling contracts

**Definition 9.7 (concrete contract).** Call *concrete* scheduling contract (or *concrete* contract) a pair \( C = (A, G) \) of composable concrete scheduling components called its *assumptions* and *guarantees*. □

To contrast with concrete contracts, we will sometimes call *abstract contracts* the scheduling contracts of Definition 9.6. The mapping from concrete to abstract scheduling components developed in Section 9.2.3 allows mapping concrete scheduling contracts to abstract ones:

\[
C = (A, G) \quad \mapsto \quad C(\overline{A}, \overline{G}) = \text{def} (\overline{A} \times \overline{G}) \quad (9.28)
\]

Say that \( C \) is *consistent, compatible, schedulable, or in saturated form*, if so is \( C(\overline{A}, \overline{G}) \). Regarding schedulability, the following holds:

**Lemma 9.10.** If \( C \) is schedulable, then it has \( G \) as a concrete implementation that remains schedulable (in the sense of Definition 9.2) when put in the context of \( A \). □

**Proof.** By Definition 9.6 regarding schedulability of abstract contracts, we have \( \overline{A} \times \overline{G} = \overline{A} \times \overline{G} \neq \emptyset \). By Lemma 9.8, we have \( \overline{A} \times \overline{G} = \overline{A \times G} \). By Statement 1 of Lemma 9.6, \( \overline{A \times G} \neq \emptyset \) if and only if \( \overline{A \times G} \neq \emptyset \), which is the implication stated in the lemma. □

The reason for considering mapping (9.28) is that only abstract scheduling contracts, not concrete ones, are equipped with the contract algebra. Observe that contract \( C(\overline{A}, \overline{G}) \) may not be in *saturated form*. To prove contract properties, we need a few criteria that are expressed in terms of concrete scheduling components and use only the algebra available for them.

By notational convention and unless confusion can result, we will simply denote by \( C \) the abstract contract associated to concrete contract \( C \) through the mapping (9.28).

#### 9.3.2 A toolbox of sufficient conditions using concrete contracts

These criteria will only be sufficient conditions, stronger than the verification of the same properties expressed in the true contract domain, i.e., by using abstract scheduling components (recall that the latter are not convenient for practical specification).
Lemma 9.11 (checking for implementation and environment relations). The following conditions imply that \( [E]^A \) is an environment and \( [M]^A \) is an implementation for \( \mathcal{C}_{(A,G)} \):

\[
E \subseteq A \quad \text{and} \quad M \text{ is composable with } A \text{ and } A \times M \subseteq G \tag{9.29}
\]

Proof. This follows from Lemmas 9.7 and 9.8. \( \square \)

Lemma 9.12 (checking for contract refinement). The following conditions imply refinement \( \mathcal{C}_{(A,G)} \leq \left( \land_{j \in J} \mathcal{C}_{(A_j,G_j)} \right) \):

\[
\forall j \in J \implies A_j \text{ is composable with } G \text{ and } \begin{cases} A_j \times G \subseteq G_j \\ A_j \subseteq A \end{cases} \tag{9.30}
\]

Proof. That (9.30) implies the desired refinement: by Lemmas 9.7 and 9.8, Condition (9.30) implies its abstract counterpart:

\[
\forall j \in J \implies \begin{cases} [A_j]^A \times [G]^A \subseteq [G_j]^A \\ [A_j]^A \subseteq [A]^A \end{cases} \tag{9.31}
\]

Focus first on environments. Pick any abstract scheduling component \( E \) such that \( E \subseteq [A_j]^A \) for some \( j \). Using the second inclusion of (9.31) we deduce that \( E \subseteq [A]^A \). Consider next implementations. Pick any abstract scheduling component \( M \) such that \( [A]^A \times M \subseteq [G]^A \). Since \( [A_j]^A \subseteq [A]^A \), we deduce \( [A_j]^A \times M \subseteq [A_j]^A \times [G]^A \), and thus (9.31) implies \( [A_j]^A \times M \subseteq [G_j]^A \). This proves that Conditions (9.31) imply the desired refinement, hence so do Conditions (9.30). \( \square \)

Formula (9.30) supports refinement checking for pairs of concrete assumptions and guarantees that do not induce contracts in saturated form.

Checking for contract refinement and composition: For \( (A,G) \) a concrete system-level contract and \((A_i,G_i), i \in I\) a set of concrete sub-contracts assigned to each subsystem in an architectural decomposition of the global system, we typically want to check if refinement \( \otimes_{i \in I} \mathcal{C}_{(A_i,G_i)} \leq \mathcal{C}_{(A,G)} \) holds. Unfortunately, checking this requires having these contracts in saturated form, see Section 5. To this end we can use Lemma 9.9, which provides a concrete formula for representing the saturated form of a contract, assuming that assumptions of this contract have no resource assigned to them. If this is not
9.3. Scheduling contracts

possible (e.g., because assumptions involve resources), we will need to work
directly in the domain of (abstract) scheduling contracts and use the approach
by Moore Interfaces, see Chapter 6.

Comment 9.13 (verification vs. synthesis of contracts). We now have the
material at hand for: 1) verifying that successive refinement steps proposed
by the designer are correct and, 2) checking for implementation and environ-
ment relations. We can also synthesize the conjunction and composition of
abstract contracts, but we have no way to reverse engineer the results back
to concrete scheduling components. To summarize, our contract framework
supports verification of independent designs but is not powerful enough to
synthesize them — as it was for instance performed for the parking garage
eexample of Chapter 10.

9.3.3 Dealing with non saturated contracts

The operation of contract saturation is algorithmically complex since it re-
quires taking complements. In this section we discuss direct sufficient con-
ditions to support independent development without the need for saturating
contracts.

Lemma 9.14. Let $\mathcal{C} = (A, G)$ be a (possibly nonsaturated) contract such
that $A, G \subseteq D^* \cup D^{\omega}$, where $X$ is some set of variables with domain $D_x$ and
$D = \prod_{x \in X} D_x$. Assume $X = X_1 \cup X_2$ and set $D_i = \prod_{x \in X_i} D_x$. Assume a
decomposition

$$G = G_1 \cap G_2$$

(9.32)

where $G_i = \text{pr}_{D \rightarrow D_i}^{-1} (H_i)$ for some $H_i \subseteq D_i^* \cup D_i^{\omega}$, meaning that $G_i$ involves
only variables belonging to $X_i$. Then, the following conditions on the pair
$(A_1, A_2)$ ensure that the two contracts $\mathcal{C}_1 = (A_1, G_1)$ and $\mathcal{C}_2 = (A_2, G_2)$ satisfy
$\mathcal{C}_1 \otimes \mathcal{C}_2 \preceq \mathcal{C}$:

$$
\begin{align*}
A_1 \cup A_2 & \supseteq A \cap \neg G \\
\neg G_1 \cup A_2 & \supseteq A \cap \neg G \\
A_1 \cup \neg G_2 & \supseteq A \cap \neg G \\
(A_1 \cap A_2) \cup \neg G & \supseteq A
\end{align*}
$$

(9.33)
How to satisfy (9.32) in practice: Suppose that the set of behaviors $G$ is defined by some finite set $E$ of equations involving the variables of $X$. We associate to $E$ its incidence graph $G_E$, which is a non directed bipartite graph with $E \cup X$ as set of vertices. $G_E$ has a branch $(x, E) \in X \times E$ if and only if equation $E$ involves the variable $x$. Recalling that $X = X_1 \cup X_2$, set $E_1 = \{ E \mid (x, E) \in G_E \text{ and } x \in X_1 \}$ and similarly for $E_2$. We have $E = E_1 \cup E_2$ (the two subsets overlap in general), which induces $G = G_1 \cap G_2$.

Proof. of the lemma. None of the considered contracts is saturated. So we first saturate them, that is, we redefine $C = (A, (G \cup \neg A))$ and $C_i = (A_i, (G_i \cup \neg A_i))$. Setting $G' = \text{def} \ G \cup \neg A$ and $G'_i = \text{def} \ G_i \cup \neg A_i$, we have, regarding the guarantees:

$$G'_1 \cap G'_2 \subseteq (G_1 \cup \neg A_1) \cap (G_2 \cup \neg A_2)$$

(by the first three lines of (9.33)) \subseteq G \cup \neg A$$

which implies $G'_1 \cap G'_2 \subseteq G'$. Regarding the assumptions, we have:

$$(A_1 \cap A_2) \cup \neg (G'_1 \cap G'_2) \supseteq (A_1 \cap A_2) \cup \neg (G \cup \neg A)$$

$$= (A_1 \cap A_2) \cup (\neg G \cap A)$$

(by the last line of (9.33)) \supseteq A$$

This shows the lemma. ☐

9.4 Sub-contracting in the development process

In this section we develop techniques in support of the following design steps:

Process 1 (development process).

1. Start with a top-level, system wide, contract. At this level, only functions are considered while computing resources are ignored. Functions are abstracted as systems of tasks with their precedence constraints. The top-level contract may be the conjunction of several viewpoints, and/or it may be specified by means of requirement tables.

2. To prepare for subcontracting to different suppliers, decompose this functional top-level contract into functional sub-contracts. So far computing resources are not considered.
3. At this step the computing infrastructure is now taken into account. Perform system wide (global) task scheduling, thus inferring resource budgets.

4. Derive resource aware sub-contracts and submit them to the supplier, for implementation. (The supplier may request a negotiation in case resource budgeting is too tight for him to meet the sub-contract.) □

This process is rather informal. It is thus tempting to interpret the above tasks as refinement steps, for scheduling contracts. With this in mind, Steps 1 and 2 exhibit no particular difficulty. Step 3, however, raises a problem. Adding the consideration of resources to a resourceless contract cannot be a refinement step. This can be seen from Lemma 9.7, which gives sufficient conditions for concrete contract refinement: referring to this lemma, there is no way that the resulting contracts \( C' = (A', G') \) can refine \( C \) since \( \|A'\|^A \supseteq \|A\|^A \) is not possible when resources are added, from \( A \) to \( A' \). This is no surprise in fact, since one cannot independently add \emph{shared} resources to different contracts, and at the same time expect to be able to develop and implement them independently.

Of course, from a theoretical standpoint, there is an easy solution to this problem. One could argue that not considering resources and budgeting them from the very beginning is a mistake and cannot work. Following this argument we would need to consider resources already in the top-level contracts, and address budgeting right from the beginning. Unfortunately, this is in total disagreement with common design practices advocating at early stages the specification of a software architecture in a resource agnostic manner.

To overcome this difficulty, our approach consists in: 1) precisely characterizing the “illegal” development steps we perform that violate contract refinement, and 2) precisely identifying the resulting risks for later system integration. To this end, we will need the following notion of “port-refinement” for concrete contracts, which is an weakening of refinement in which only ports are taken into consideration.

**Port-refinement of contracts:** Decompose the alphabet \( \mathcal{V} \) introduced in Definition 9.4:

\[
\mathcal{V} = (\{0, 1\})^P \times (\prod_{r \in R} \Sigma_r) = \mathcal{V}^P \times \mathcal{V}^R
\] (9.34)
For \(M = ((T, R, \rho), L)\) a concrete scheduling component, define
\[
[M] = \text{def } \text{pr}_{\forall \rho}([M_{\rho/e}])^A,
\]
where \(M_{\rho/e} = \text{def } ((T, \emptyset, \varepsilon), L),\) (9.35)
and \(\varepsilon\) is the allocation map with empty domain. In words, we first ignore
the possible conflicts due to shared resources (replacing \(M\) by \(M_{\rho/e}\)), we
then take the abstract semantics \([M_{\rho/e}]^A\), and we finally project the result-
ing abstract semantics over the ports only (taking \(\text{pr}_{\forall \rho}(...))\). \([M]^P\) captures
the scheduling aspect of \(M\) while discarding the resource aspect of it. Observe that \([M]^P\)
contains the language obtained by projecting \([M]^A\) over the ports; this inclusion is generally strict. If, however, \(M = ((T, \emptyset, \varepsilon), M)\) is
resourceless, then \([M]^P = \text{pr}_{\forall \rho}([M]^A).\) For \(C = (A, G)\) a concrete schedul-
ing contract, define
\[
[C]^P = \text{def } ([A]^P, [G]^P),
\]
the port-contract associated with \(C).\) (9.36)
Despite the boldface notation used, port-contracts are abstract contracts. For \(C\) and \(C'\) two concrete contracts, say that
\(C'\) port-refines \(C\), written \(C' \preceq P C\) if \([C']^P \leq [C]^P).\) (9.37)

\(T\)-closed contracts and illegal development steps: We restrict these
steps to the following situation. Assume, from early design stages on, prior
knowledge of the following property about a given set \(T\) of tasks — this does
not require detailed knowledge of the computing resources:

Definition 9.8 \((T\)-closed contracts\). Say that a set \(T \subset T\) of tasks is segre-
gated if the set \(R\) of all resources partitions as follows:
\[
R = R_T \cup R_T, R_T \cap R_T = \emptyset, \text{ and } \left\{ \begin{array}{l}
T \subseteq \Sigma_{R_T} \\
T - T \subseteq \Sigma_{R_T}
\end{array} \right.
\]
(9.38)
For any segregated set of tasks \(T\), say that concrete contract \(C = (A, G)\) is
\(T\)-closed if \(T_G \subseteq T\) and \(T_A \cap T = \emptyset.\) \(\Box\)

An instance of \(T\)-closed set of tasks will naturally occur in our application
case. If \(C = (A, G)\) is \(T\)-closed, then \(\rho_G(T_G) \cap \rho_A(T_A) = \emptyset\) holds. Illegal steps
are performed on \(T\)-closed contracts only. An illegal step consists in replacing
\(T\)-closed contract \(C\) by another \(T\)-closed contract \(C'\) port-refining it: \(C' \preceq P C.\)
The resulting risks at system integration: Port-refinement being not a refinement, replacing $C$ by $C'$ will not ensure that any implementation of $C'$ will meet the guarantees of $C$ under any legal environment for $C'$ — it should ensure this if it was a true refinement. Still, the following result holds, which precisely bounds the risks at system integration. In this lemma, we generically denote by $M$ the abstract scheduling component associated to $M$.

**Theorem 9.15.** Let be $C' \preceq_p C$ satisfying the following conditions:

1. $C$ and $C'$ are $T$-closed for a same segregated set $T$ of tasks,
2. $C'$ is schedulable,
3. $\llbracket A' \rrbracket^P = \llbracket A \rrbracket^P$, $A$ and $A'$ both have their tasks pairwise non-conflicting, and
4. $G$ is resourceless.

Then, the following holds: $\emptyset \neq A \times G' \sqsubseteq G$. □

**Proof.** Property $\emptyset \neq A \times G'$ follows from Condition 3 and the assumption that $C'$ is schedulable. Finally, since $C' \preceq_p C$, we have $\llbracket A \rrbracket^P \times \llbracket G' \rrbracket^P \sqsubseteq \llbracket G \rrbracket^P$, which implies $A \times G' \sqsubseteq G$ since $G$ is resourceless. □

**Comment 9.16.** Theorem 9.15 expresses that $G'$ is an implementation of $C'$ satisfying the following property: when put in the context of the most permissive environment of $C$, meets the guarantee $G$ and is schedulable. That $G$ is met will remain valid for any legal environment of $C$ and any implementation of $C'$. Schedulability, however, is only ensured by the most permissive environment of $C$ and implementation of $C'$. This restriction is not surprising since schedulability is a liveness property whereas A/G-contracts support only safety properties. □

**The development process 1 made safe:** We are now ready to explain how the development process 1 can be made safe by implementing the illegal development steps safely.

**Process 2** (development process 1 made safe). We assume a segregated subset $T$ of tasks.
1. Start with a top-level, $T$-closed, contract $C_{\text{top}}^{\text{func}} = (A_{\text{top}}, G_{\text{top}})$. At this level, only functions are considered while computing resources are ignored. Functions are abstracted as systems of tasks with their precedence constraints. The top-level contract may be the conjunction of several viewpoints, and/or it may be specified by means of requirement tables.

- **Comment:** No change w.r.t. Process 1 besides $T$-closedness.

2. To prepare for subcontracting to different suppliers, decompose the above functional contract $C_{\text{top}}^{\text{func}}$ into functional, resource agnostic, sub-contracts in such a way that

$$C_{\text{func}}^{\text{ref}} = (A_{\text{ref}}, G_{\text{ref}}) = \times_{i \in I} C_i \text{ satisfies } \begin{cases} \forall_{i \in I} A_{\text{top}} \not\sqsubseteq G_{\text{ref}} \sqsubseteq G_{\text{top}} \\ A_{\text{ref}} \sqsubseteq A_{\text{top}} \end{cases} \quad (9.39)$$

where the $C_i$ are $T$-closed subcontracts for the different suppliers. In addition, we require that $A_{\text{ref}}$ and $A_{\text{top}}$ possess identical sets of tasks, i.e., map $\psi$ of Lemma 9.7 is the identity. By Lemma 9.12, (9.39) ensures $\psi_{\text{func}}^{\text{ref}} \leq \psi_{\text{func}}^{\text{top}}$. So far resources were not considered.

- **Comment:** No change so far, with respect to Process 1, besides naming contracts and making refinement step precise through (9.39). The first two steps make no reference to semantics, meaning that no scheduling analysis is required, cf. Comment 9.3. From the next step on, this process deviates from Process 1.

3. At this step the computing resources are now taken into account. Allocate a resource to each task of $A_{\text{ref}}$ and $G_{\text{ref}}$, in such a way that all tasks of $A_{\text{ref}}$ are pairwise non-conflicting, see Definition 9.1. Precedence constraints between tasks are not modified. This yields a resource aware $T$-closed contract $C_{\text{res}}^{\text{ref}}$ such that

$$C_{\text{res}}^{\text{ref}} = (A_{\text{ref}}^{\text{res}}, G_{\text{ref}}^{\text{res}}) \leq_p C_{\text{func}}^{\text{ref}} \quad (9.40)$$

Since $A_{\text{ref}}^{\text{res}}$ is free of conflict, only $G_{\text{ref}}^{\text{res}}$ requires a non-trivial scheduling analysis, which result is specified through the semantics $\llbracket G_{\text{ref}}^{\text{res}} \rrbracket$, see Comment 9.3. At this point, resources have been globally budgeted and scheduling analysis globally performed.
9.5. Modeling methodology

In this section we discuss the use of our framework in practice. We first discuss how scheduling components capture scheduling problems in practice. Then, in order to capture read and write actions we propose an extension of our existing set of pure events carried by ports.

9.5.1 Capturing scheduling problems with our framework

Scheduling problems are captured by our notion of Concrete Scheduling Component $M=(K,L)$. Sort $K$ identifies the set of tasks together with their ports, induced precedence conditions, and the resource allocation map. Language $L$ allows expressing various dynamical constraints such as, for instance:

1. Bounds on the execution time of tasks, i.e., the number of busy slots for each epoch;
2. Bounds on the duration of intervals $[t, c]$, from start to completion events;
3. Bounds on the intervals $[p(t), t]$ from release times to start times of tasks;
4. Minimum inter-arrival time between two successive triggers $p(t)$ for a task;
5. Bounds on the response time interval $[p(t), p(t)]$ of tasks for each output $p(t) \in P(t)$ produced by the task (this captures deadlines);
6. When combining the consideration of $K$ and $L$ it is possible, for two tasks $\tau_1$ and $\tau_2$ such that $\tau_1 \leq \tau_2$, to express in $L$ end-to-end bounds for the interval $[t, c]$. 

• Comment: This is the illegal step, protected by Theorem 9.15.

4. Continue by decomposing contract $C_{\text{res}}$ into resource aware sub-contracts $C_{i}^{\text{res}}$, following the architecture specified at Step 2, in such a way that $\otimes_{i \in I} C_{i}^{\text{res}} \leq C_{\text{ref}}^{\text{res}}$. The results of the next section can be used for this. □
Observe that bounds 1–3 cannot be expressed using abstract scheduling components, but other above listed properties 4–6 can.

In the literature about real-time scheduling analysis [189, 69, 238], there exist a common understanding about typical task parameters, which are important for 1) defining timing constraints and 2) defining algorithms solving scheduling problems like checking feasibility of a schedule or the schedulability of a task set. Figure 9.5 puts these task parameters in the context of our model of concrete scheduling components. For a task $\tau$, the following parameters are typically of interest:

- $r$: The release or activation time of a task. This is the point in time when a new job of a task is created and becomes ready for execution.

- $s$: The start time of a task. This is the point in time when the task starts executing its job. It may coincide with the release time or delayed for example by a higher priority task that is executing.

- $f$: The finishing time of a task. This is the point in time when the task finished its current job and terminates.

- $R$: The response time of a task. This is the difference between the finishing time the release time of a task: $R = f - r$.

- $d$: A relative deadline of a task. This is the point in time when the task must have finished its current job. Usually this is seen relative to $r$. A task meets its deadline if $R \leq d$.

- $e$: The execution time of a task. This is the time a task needs for execution on its processor without being preempted.
So far we discussed the specification of monolithic scheduling problems. Compositionality is supported by contracts, which we discuss next.

Our model of Definitions 9.1 and 9.2 carries the essence of real-time scheduling, namely: resources, tasks, precedence conditions between tasks, and the language \( L \) that can be used to express various timing assumptions or guarantees. Now, this model does not offer the expressiveness we need for our application case developed in Section 11.

### 9.5.2 Extensions used in practice

To avoid complicating our theoretical development, we only give informal explanations. So far in our current framework of Definitions 9.1 and 9.2, each task \( \tau \) possesses one trigger and one completion port. In fact, our practical application requires considering additional ports.

Figure 9.6 motivates the extension with regard to reads/writes of variables. Identification of activation or input ports with output or completion ports of other tasks represent data dependencies within a task set. In [189] such dependencies are discussed in terms of a task graph, which is distinguished from a precedence graph, with the former being an extension of the precedence graph by adding different kinds of edges between tasks denoting for example data dependencies. Such data paths may involve under-and oversampling effects like depicted in Figure 9.6. Though these dependencies do not necessarily coincide with task precendences, it is typical in control engineering to require input values to not exceed a certain age (called AgeConstraint in [115]). At the same time, data loss due to undersampling is acceptable. This discussion motivates the following extensions related to ports:

**Input and Output ports:** In addition to its trigger and completion ports \( p'(\tau) \) and \( p^c(\tau) \), each task \( \tau \) possesses two sets of input ports \( P^i(\tau) = \{ p^i(\tau, k) \mid k \in K_\tau \} \) and output ports \( P^o(\tau) = \{ p^o(\tau, \ell) \mid \ell \in L_\tau \} \). As part of the specification of the considered concrete scheduling component \( M \), the relation \( \Rightarrow \) between ports introduced in (9.2) can be extended to the whole set of ports associated to task \( \tau \), namely \( \{ p'(\tau) \} \cup P^i(\tau) \cup P^o(\tau) \cup \{ p^c(\tau) \} \), with the condition that the trigger port and completion port remain minimal and maximal in this extended order. Since ports can be shared between tasks, the precedence
relation \( \rightarrow \) extends to the set of all ports of the concrete scheduling component \( M \). The reason for this extension is that tasks might have several control points from which certain “posterior” tasks can be launched.

**Read and Write actions:** To each input port \( p^i(\tau, k) \) we associate a read action occurring at some busy slot of the task. Similarly, to each output port \( p^o(\tau, \ell) \) we associate a write action occurring at some busy slot of the task.

**Semantics:** The semantics \([M]\) is obtained by adapting Definition 9.2 to the refined ordering between start and completion of tasks and the various read and write actions, and meeting the conflict freeness condition.

**Extending Lemma 9.7 regarding inclusion:** In this lemma tasks get refined by using the surjective total map \( \psi : T_1 \rightarrow T_2 \), where \( T_1 \) is the set of tasks of the refined concrete scheduling component. Since the pair of trigger and completion ports, which was associated to each task, is now replaced by a pair of sets of ports, we need to replace the map \( \psi \) acting on tasks by a surjective map \( \Psi : P_1 \rightarrow P_2 \) acting on ports. As a counterpart of Condition 1 of Lemma 9.7, we require that \( \Psi(\leq_1) = \leq_2 \), where \( \leq_i \) is the precedence order on \( P_i \), for \( i = 1, 2 \). Conditions 1b and 9.19 of Lemma 9.7 remain unchanged.

### 9.6 Bibliographical note

The following text is quoted verbatim from the pioneering work by Insup Lee et al. [241]: “Real-time systems could benefit from component-based design, only if components can be assembled without violating compositionality on timing properties. When the timing properties of components can be
analyzed compositionally, component-based real-time systems allow components to be developed and validated independently and to be assembled together without global validation.” The reader is referred to this paper for earlier related work from the real-time scheduling community. This paper develops a model of scheduling interface collecting the workloads, resources, and scheduling policy, addressing the above quoted objectives. Concepts and techniques used originate from the real-time scheduling community. Specific classes of hard real-time system scheduling problems are considered, namely periodic models and bounded-delay models. This group of authors has further developed the same track with the same techniques, enlarging the classes of real-time scheduling problems considered. This significant body of work is nicely summarized in the tutorial paper [16] and implemented through the CARTS tool for compositional analysis of real-time systems [219]. One interesting application case concerns the scheduling of ARINC partitions [120]. The approach generally followed in these works is the following. The schedulability problem is structured as a hierarchy of subproblems. The solution of each subproblem is summarized using some form of interface (depending on the particular approach) and, at the next upper level, the amount of additional resource and deadline conditions is computed as the solution of some optimization problem.

Lothar Thiele and co-workers have developed for real-time scheduling an elegant algebraic framework called the Real-Time Calculus (RTC) [249, 254, 199]. This algebraic framework builds on top of the foundational work on max-plus algebra, initially developed in the formerly available book Synchronization and Linearity, (1992), by F. Baccelli, G. Cohen, G. J. Olsder and J. P. Quadrat. This framework was developed to endow the class of event graphs (the subclass of conflict free Petri nets) with an algebra of linear input-output transfer functions. Components of the RT Calculus are thus linear transfer functions in this max-plus algebra and interface behaviors are expressed as arrival curves, which specify lower and upper bounds for event arrivals. Our Figure 9.1 could indeed be interpreted in this way, where wires carry flows of events driven by this algebra — in fact, our mapping from concrete to abstract scheduling components (Definition 9.5) is a trick for representing Thiele’s max-plus algebra by a dataflow composition. Reference [249] considers real-time interfaces where assumptions and guarantees are expressed by means of
arrival curves on inputs and outputs, respectively. Refinement of such interfaces is characterized and a parallel composition is defined. So-called adaptive interfaces are proposed in which arrival curves are propagated throughout the network of components, compositionally. This topic is further developed in [247]. A blending of this model with timed automata is studied in [172] together with a mapping of RTC-based real-time interfaces to timed automata. The very interesting work [190] applies and develops similar techniques for distributed heterogeneous time-triggered automotive systems.

Our work in this section has its roots in [246, 231, 232, 245]. Our aim is different from the previous set of references and complements it nicely. Whereas previous references considered restricted classes of real-time scheduling problems and addressed them with complete algorithmic solutions, our model makes no restriction on the class of scheduling problems, except that the allocation of tasks to resources is static and so are precedence conditions. In turn, we provide a full fledged contract algebra decomposing system wide problems into an architecture of (smaller) local scheduling problems, seen as “proof obligations”, but, we do not discuss how these proof obligations can be automatically or algorithmically checked. These proof obligations can be either delegated to existing real-time scheduling algorithms (see the previous references) or addressed manually, or by using a model checking engine for a restricted class of scheduling contracts. That is, we favor generality over full automatization — still, the lifting of local proofs to system-wide solutions is supported by our contract algebra. In doing so we specifically target OEM-supplier relations in a supplier chain, by providing support for decomposing a system-level scheduling contract into sub-contracts for suppliers while guaranteeing safe system integration, and support for fusing different viewpoints on the system using contract conjunction — either relevant to schedulability analysis or to different aspects of the system. With comparison to the above four references [246, 231, 232, 245], our development of Section 9.2 fully matches the meta-theory of contracts of Chapter 4. This way, we inherit refinement and parallel composition (like some previous real-time interface models offer), and also a conjunction, which allows to specify real-time scheduling problems in a “requirement engineering” style.

We would like to conclude this discussion by the following question the reader may want to ask: did we really reuse the material of theory oriented
9.6. Bibliographical note

Chapters? Can we really claim that the theory developed there has a wide applicability? The point is that the development made in Section 9.2 cannot be claimed trivial: is it always the case? Here are our answers: First, the reader should not underestimate the cost of upgrading a framework of components (offering a $\times$ but no refinement nor conjunction) to a corresponding framework of contracts — the theory oriented sections illustrate how much it can cost. Second, and most importantly, our solid foundations provide very precise guidelines regarding what a framework of components should offer. These guidelines were indeed extremely useful in developing our model of scheduling components.
10

Contracts for Requirement Engineering

10.1 Motivation: formalizing requirements

Requirements are nowadays typically expressed in the form of Doors\textsuperscript{1} sheets. Such sheets combine informal text statements with or without figures, formal or informal models, or semi-formal or formal statements from various specification languages. Requirements are the means by which an OEM interacts with its supplier chain, on both a legal and a technical perspective. For this reason, requirement engineering is the area of choice for contract based design, so we devote to this design activity a special section.

What is the intended meaning of requirements documents? Requirements generally aim at specifying the guarantees $G_1, \ldots, G_k$ that are expected from the system. Defining these guarantees well is the primary focus of requirement management. Guarantees, however, generally go along with respective sets of assumptions $\{A_{1}^{1}, \ldots, A_{\ell_{1}}^{1}\}, \ldots, \{A_{1}^{k}, \ldots, A_{\ell_{k}}^{k}\}$ expressing boundary conditions or legal contexts of use. Such assumptions are most often left implicit, which is both a source of problems at system integration, and a source of dispute regarding liability between the OEM and its suppliers in case a problem occurs.

\textsuperscript{1}https://www.ibm.com/us-en/marketplace/rational-doors
10.1. Motivation: formalizing requirements

Even if assumptions are carefully listed, no clear difference is generally made between how assumptions combine, versus how guarantees combine, in a requirements document. Clearly, guarantees must combine in a conjunctive way. This is indeed reflected by the common practice that “the system must pass all tests attached to the different requirements”. What about assumptions? If the system is used in a way that violates some assumption, then the system is relieved from the set of guarantees that relied on this assumption. Other guarantees, however, remain. In fact, each guarantee $G$ should be considered along with its associated set $\{A_1, \ldots, A_\ell\}$ of assumptions and interpreted as the entailment

$$\{A_1, \ldots, A_\ell\} \Rightarrow G.$$  \hfill (10.1)

The right interpretation of a requirement document is thus the conjunction of all such entailments — clearly, assumptions must not be conjuncted. Contract theories offer a notion of **conjunction** that interprets requirements documents in the above way. This very same concept is also valid to set the meaning of how the combination of different chapters or **viewpoints** of the systems requirements must be interpreted. Typical instances of viewpoints are function, safety, energy, etc. These viewpoints rely on different modeling frameworks but nevertheless generally interact. This calls for supporting heterogeneous modeling in contract based design. Being generic, the meta-theory of contracts we have developed in Chapter 4 is a useful step toward supporting heterogeneity.

How to deal with subsystems requirements for suppliers? Once the OEM has its system-level requirements at hand, it proceeds to defining the sets of requirements attached to the different subsystems it has identified in its architectural study. The natural question is then: does the **composition** of these subsystems requirements documents properly **refine** the system level requirements? The two words in italics are two concepts that must be properly clarified. Once more, there must be a difference in handling assumptions and guarantees while composing requirements documents attached to different subsystems. The conjunctive interpretation behind the statement “all tests must be passed” is clearly superficial and indeed erroneous. It is acknowledged by skilled designers that part of the assumptions for a subsystem may be discharged by the guarantees offered by other subsystems. Unfortunately, this is not easily reflected in a simple aggregation of some requirements. How
the subsystem’s assumptions are discharged by the other subsystems actually requires the notion of contract composition extensively studied in the metatheory of Chapter 4. In the same vein, assumptions and guarantees are handled differently in the notion of contract refinement, which again rules out the naive “all tests must be passed” discipline when confronting subsystems requirements to system level requirements.

In this section, we discuss the use of contracts for requirement engineering. We illustrate our approach by means of a small example, a car parking system, that is representative of early requirements capture. The following issues arise in requirement capture. For each of them, we briefly indicate how our example illustrates them:

1. The top-level system specification is captured in a table or document collecting different kinds of requirements expressed using different formalisms (constrained natural language, boilerplate requirements [22], all sorts of automata theoretic formalisms, scenario languages, logics). Different formalisms may be used for different kinds of requirements in a same specification.

   In our example, we illustrate this by blending textual requirements written in constrained English with tiny automata, expressing elementary behavioural properties.

2. The requirements document is often structured into chapters describing various aspects of the system.

   In our example, there is a clean separation between the specification of how a gate should behave, and of how a payment subsystem should proceed.

3. The current practice is that assumptions are often implicit, and even when they are explicitly stated, the pairing with the guarantees is missing. As argued before, requirement documents should be structured in such a way that each guarantee is paired with a subset of assumptions explaining the operational context under which this guarantee is expected to hold.
10.2. The car parking system, informal presentation

We begin with the informal presentation of the example. We first state the top-level requirements. Then, we discuss how to map the top-level requirements to an architecture of sub-contracts attached to the different sub-systems. The aim is that each sub-system is developed independently by a different supplier, while safe system integration is ensured by construction.

10.2.1 Top-level requirements

The system under specification is a car parking subject to payment of a fee before exiting. At its most abstract level, the requirements document comprises three chapters: gate, payment, and supervisor, see Figure 10.1. The gate chapter collects the common requirements regarding one or several entry and exit gates. These common requirements will then be instantiated to entry and exit gates.

Focus on the "gate" chapter. It consists of the three requirements shown on Figure 10.1. Requirement $R_{g,1}$ can be described by means of

\[
\text{forall } g \in \{\text{entry}, \text{exit}\} \\
R_{g,1}: \text{"vehicles shall not pass when gate is closed", see Fig. 10.2} \\
R_{g,2}: \text{"vehicle_pass is forbidden next to vehicle_pass} \\
R_{g,3}: \text{!gate\_open is forbidden next to !gate\_open, and} \\
\text{!gate\_close is forbidden next to !gate\_close} \\
(R_{g,1}, R_{g,2}) \rightarrow R_{g,3}
\]
an i/o-automaton, as shown in Figure 10.2 — in Figure 10.1, we only provide a specification in natural language. Suppose that some requirement says: "?gate_open never occurs". This is expressed by having no mention of ?gate_open in the corresponding i/o-automaton — this way of doing assumes that the alphabet of actions of the i/o-automaton is explicitly given. The other two requirements are written using constrained natural language, which can be seen as a boilerplate style of specification. Prefix "?" indicates an input and prefix "!" indicates an output.

The first two requirements are not under the responsibility of the system, since they rather concern the car driver. Thus it does not make sense to include them as part of the guarantees offered by the system. Should we remove them? This would be problematic. If drivers behave the wrong way unexpected things can occur for sure. The conclusion is that 1) we should keep requirements R_{g.1} and R_{g.2}, and 2) we should handle them differently than R_{g.3}, which is a guarantee offered by the system. Indeed, R_{g.1} and R_{g.2} can only be seen as assumptions and R_{g.3} is the guarantee stated under the operational context of R_{g.1} and R_{g.2}. This pairing between a guarantee and its related assumptions is denoted using the graph notation in the last line of Table 10.2. Such a pairing can be easily implemented as hyperlinks in requirement engineering tools. So far we have specified gate as a list of requirements. Requirement R_{g.1} specified as an i/o-automaton can be considered formal. Requirements R_{g.2} and R_{g.3} are formulated in constrained natural language and are ready for subsequent formalization (e.g., using i/o-automata too).

So far we discussed one particular aspect of the system (its generic gate). We need to review all the aspects of the system (i.e., also the payment and the supervisor). Then, we must formalize what top-level specification results
from “merging” the specifications of all the aspects. Informally,
\[ C = \text{“merge”}(C_{\text{entry\_gate}}, C_{\text{exit\_gate}}, C_{\text{payment}}, C_{\text{supervisor}}). \]

The specification for the merge operation is that, if we restrict the alphabet of actions of the global system to one among the sub-alphabets attached to entry_ or exit_gate, payment, or supervisor, then the corresponding chapter should be satisfied. Thus, the overall system must satisfy all the aspects of the top-level specification. From the contract meta-theory developed in Chapter 4, the formalization of this merge operation is by taking the conjunction of the contracts attached to the different aspects of the system:
\[ C = C_{\text{entry\_gate}} \land C_{\text{exit\_gate}} \land C_{\text{payment}} \land C_{\text{supervisor}}. \] (10.2)

In addition, we have to check if
- these contracts are consistent, i.e., non-contradicting — for this we can offer for most classes of contracts automatic methods checking consistency;
- this specification is complete, i.e., it defines for all possible environments a precise system reaction — for this, we offer methods such as playing out requirements, following ideas by D. Harel in [149, 147].

The formalization of these requirements as contracts, as well as methods to check the consistency and completeness of those contracts, will be developed in Section 10.3.4.

We insist that, at the top-level specification for this parking example, we consider system aspects or viewpoints, not sub-systems. We have considered that payment, gates, and supervisor, are such aspects. Other aspects for consideration may typically be: the function, the safety, the timing, the resources — we did not consider such aspects in our example because timing and quantitative resources are beyond the capability of the MICA tool we developed. More flexible processes for requirement capture are discussed in Section 10.4.

**10.2.2 Sub-contracting**

So far sub-systems and system architectures were not considered. Having the top-level specification \( C \) at hand, the designer then specifies an architecture
à la SysML, as shown on Figure 10.3. Some comments are in order regarding this architecture.

![System architecture diagram]

**Figure 10.3:** System architecture as specified by the designer.

The considered instance of car parking system consists of one entry gate, one exit gate, and one payment machine. Compare with the top-level specification (10.2). The latter comprises an entry- and an exit-gate, a payment machine, and a supervisor, each one with its set of requirements. In contrast, the architecture of Figure 10.3 involves no supervisor — the latter is meant to be distributed among the two gates. So the system architecture does not match the structure of the top-level specification — this is a typical situation encountered in practice.

The next step in the design consists in sub-contracting the development of each of the three sub-systems of the architecture of Figure 10.3. This amounts to specifying three sub-contracts $C_{\text{EntryGate}}$, $C_{\text{ExitGate}}$, and $C_{\text{PaymentMachine}}$, such that, using the notations of Chapter 4:

$$C_{\text{EntryGate}} \otimes C_{\text{ExitGate}} \otimes C_{\text{PaymentMachine}} \preceq C$$

Recall that refinement $\preceq$ in (10.3) means that any implementation of the left hand side is also a valid implementation of the top-level $C$ and any legal operational use (we call it: environment) of $C$ is also legal for the left hand side.
Then, the contract composition operator $\otimes$ ensures that each supplier can develop its sub-system based on its own sub-contract only, and, still, integrating the so designed sub-systems yields a correct implementation of the top-level specification.

10.3 Formalization using contracts

Despite being simple and small, the car parking system example quickly becomes complex for reasons that are intrinsic to the formal management of requirements. The MICA\textsuperscript{2} tool was used to develop it [70]. In this development, requirements were written in constrained English language and then translated into Modal Interfaces — while the presented translation is manual, automatic translation could be envisioned.\textsuperscript{3} Once this is completed, contracts are formally defined and the apparatus of contracts can be used. In particular, important properties regarding certification can be formally defined and checked, e.g., consistency, compatibility, correctness, and completeness. In addition, support is provided for turning top-level requirements into an architecture of sub-systems, each one equipped with its own requirements. The latter can then be submitted to independent suppliers for further development. We begin with the presentation of the contract framework we will be using, namely Modal Interfaces.

10.3.1 The contract framework

We use Modal Interfaces (with variable alphabet). In words, Modal Interfaces are input/output automata whose transitions are labelled by $\textit{may}$ and $\textit{must}$ modalities. The inputs are under the control of the environment whereas the outputs are under the control of the considered system. While the system is in a given state, the environment is only allowed to submit inputs that meet a $\textit{must}$ transition of the Modal Interface at that state — this way all legal implementations of the Modal Interface will be prepared to handle such inputs. On the other hand, the system is only allowed to emit outputs that meet a $\textit{may}$

\textsuperscript{2}http://www.irisa.fr/s4/tools/mica/Mica_A_Modal_Interface_Compositional_Analysis_Library/Introduction.html

\textsuperscript{3}In fact, the contract specification languages proposed in the projects SPEEDS [40] and CESAR are examples of translations from a constrained English language to a formal models of contracts similar to Modal Interfaces.
transition of the Modal Interface at that state and it must offer all *must*-labeled transitions. A formal presentation of Modal Interfaces is found in Section 8.3. There are three main reasons for this design choice:

1. By offering the *may* and *must* modalities, Modal Interfaces are well suited to express mandatory and optional behaviors in the specification, which we consider convenient for requirement engineering.

2. Being large sets of requirements structured into chapters, requirements documents are a very fragmented style of specification. Modal Interfaces offer the needed support for an accurate translation of concepts such as “set of requirements”, “set of chapters”, together with a qualification of who is responsible for each requirement (the considered component or sub-system versus its environment).

3. At the top-level, conjunction prevails. However, as soon as the designer refines the top-level requirements into an architecture of sub-systems, composition enters the game. Turning a conjunction of top-level requirements into a composition of sub-systems specifications thus becomes a central task. Modal Interfaces provide significant assistance for this.

Overall, the problem considered in the above claim 3 can be stated as follows. The designer begins with some system-level contract \( \mathcal{C} \), which is typically specified as a conjunction of viewpoints and/or requirements. The designer guesses some topological architecture by decomposing the alphabet of actions of \( \mathcal{C} \) as

\[
\Sigma = \bigcup_{i \in I} \Sigma_i \quad , \quad \Sigma_i = \Sigma_{i_{\text{in}}} \uplus \Sigma_{i_{\text{out}}}
\]

such that composability conditions regarding inputs and outputs hold. Once this is done, we expect our contract framework to provide help in generating a decomposition of \( \mathcal{C} \) as

\[
\bigotimes_{i \in I} \mathcal{C}_i \preceq \mathcal{C}
\]

where sub-contract \( \mathcal{C}_i \) has alphabet \( \Sigma_i = \Sigma_{i_{\text{in}}} \uplus \Sigma_{i_{\text{out}}} \). Guessing architectural decomposition (10.4) relies on the designer’s understanding of the system and how it should naturally decompose — this typically is the world of SysML.
Finding decomposition (10.5) is, however, technically difficult in that it involves behaviors. The algorithmic means that were presented in Section 8.6 provide the due answer. In this car parking system, we use the special operation of restriction that was developed in that section.

10.3.2 Top-level requirements

We formalize top-level requirements using Modal Interfaces. In our formalization we assume a process for requirement capture in which the top-level specification is structured into a set of chapters or aspects, with no consideration of sub-systems and system architecture. More flexible processes are discussed in Section 10.4.

We first explain how the specification of “gate” in Figure 10.1 translates into Modal Interfaces. Observe that each requirement $R_{g,j}$ of Figure 10.1 is a sentence that can be formalized as an i/o-automaton, see Figure 10.2 for such a formalization of requirement $R_{g,1}$. Since the top-level specification is structured by aspects with no consideration of sub-systems, each (chapter of a) requirement document $D$ collects requirements (assumptions or guarantees) for which the input/output status of the different actions is consistent, meaning that no action exists in $D$ that is an input in some requirement and an output in another one. Furthermore, document $D$ takes the form of a set of causal pairs $A_i \rightarrow G_i$

$$D = \{ A_i \rightarrow G_i | G = \cup_{j \in I} G_i \text{ and } A_i \subseteq A \} \tag{10.6}$$

where the subset $G$ of guarantees of $D$ decomposes as $G = \cup_{j \in I} G_i$ and $A_i \subseteq A$ is the subset of the assumptions of $D$ on which each guaranty belonging to $G_i$ relies—note that the $A_i$’s can overlap. Referring to Figure 10.1 for gate, there is only one guarantee which requires the two assumptions to hold: $\{R_{g,1}, R_{g,2}\} = A \rightarrow G = \{R_{g,3}\}$. The translation of document $D$ specified as in (10.6) is specified by the following set of rules, where $G_i = \{R_{ij} | j \in J_i\}$ and $A_i = \{R_{ik} | k \in K_i\}$.

**Rules 1** (translating individual guarantees). For each causal pair $A_i \rightarrow G_i$, we start from a description of each guarantee $R_{ij} \in G_i$ as an i/o-automaton. This i/o-automaton is translated to a Modal Interface by applying the following rules:
**R** G1 Unless otherwise explicitly stated, transitions labeled by an output action are given a “may” modality. The rationale is that the default semantics for guarantees is best effort. The only exception is when the requirement specifies that an output action is mandatory, e.g., by having a “must” in the sentence.

**R** G2 Transitions labeled by an input action of the considered system are given a “must” modality. The rationale is that implementations may not refuse this input action in this state.

Applying these rules to **R** ij yields a modal interface called **G** ij. □

Performing this for the single guarantee **R** g3 of gate yields the Modal Interface shown in Figure 10.4. In this and the coming figures, Modal Interfaces are drawn with may transitions being dotted and must transitions being solid.

![Figure 10.4: Translating the guarantee **R** g3 of gate as an i/o-automaton (top) and then as a Modal Interface **G** gate (bottom) using Rules 1.](image)

In the following, the sentence “complementing the status input/output” indicates that the two statuses are exchanged.

**Rules 2** (translating individual assumptions). For each causal pair **A** i → **G** i, we start from a description of each assumption **R** ik ∈ **A** i as an i/o-automaton. This i/o-automaton is translated to a Modal Interface by applying the following sequence of rules:

1. We complement the status input/output in every assumption **R** ik, thus taking the point of view of the environment; we call the result **R** ↑ ik;

2. Having done this we apply Rules 1 to each **R** ↑ ik. This yields, for each **R** ↑ ik, a Modal Interface **A** ↓ ik that must be satisfied by every environment;
complementing backward the status input/output of each action yields a modal interface $A_{ik}$.

Performing this for the assumptions $R_{g,1}$ and $R_{g,2}$ of gate yields the Modal Interfaces $A_1$ and $A_2$ shown in Figure 10.5.

![Diagram](image)

**Figure 10.5:** Translating the assumptions of gate using Rules 2. Recall that dotted/solid transitions indicate may/must modalities.

**Rules 3** (combining assumptions and guarantees). The modal interface $C_i$ representing the causal pair $A_i \rightarrow G_i$ is then computed as indicated in Section 8.7. Finally, the top-level contract $C$ is the conjunction of the contracts associated to each causal pair: $C = \bigwedge_{i \in I} C(A_i, G_i)$.

Performing this for the whole chapter gate yields the Modal Interface shown in Figure 10.6.

**Comment 10.1.** Regarding the guarantees offered by the component, allowed outputs possess a may modality, which reflects that Guarantees specify what the component may deliver. Other actions are forbidden. Regarding the context of operation, legal inputs to the gate (e.g., vehicle_pass when exiting state “2”) have a must modality. This complies with the intuition that the component should not refuse legal stimuli from its environment. Violation of the contract by its environment occurs when an illegal input is submitted by the environment (vehicle_through when exiting state 0 or state 3). Hence, the whole contract gets relaxed by moving to the special state “1” from which
any action is allowed — such a state is often called a “top” state. It captures an unpredictable response to an exception input.

The same procedure applies to all chapters gate, payment, and supervisor, of the top-level textual specification, shown in Figure 10.7 (it is an expansion of Figure 10.1).

**Comment 10.2 (Requirements not supported by our formalization).** This table involves a supplementary chapter for the supervisor, written in blue, which collects requirements that are not supported by our formalization. The reason for this lack of formal support is that infinite domains of data are beyond the reasoning capabilities of the MICA tool we are using, whereas tickets are infinitely many and must be characterized by a unique identifier. As a consequence, guarantee R_{s,9} is outside the scope of our formal analysis. It is indeed a typical situation in practice, that only a subset of the requirements can be formally supported — requirements address all aspects of system specification, be they within or outside the scope of formalization. For the case of Figure 10.7, requirement R_{s,9} must be validated against the designed system, either manually, or via observer techniques.

The Modal Interfaces encoding chapters payment and supervisor of the top-level are displayed in Figures 10.8 to 10.10. Figure 10.10 showing the contract associated to the supervisor is unreadable and the reader may wonder why we decided to put it here. We indeed wanted to show that, when contract design is performed formally and carefully, top-level contracts rapidly
10.3. Formalization using contracts

Table 10.1: Top-level requirements. Assumptions and Guarantees are written in italics and roman, respectively. The last line of each chapter specifies the causal pairs following (10.6). Quoted requirements are written in natural language: the corresponding formalization as a modal interface is omitted. The additional chapter, in blue, collects requirements not supported by our formalization.

\textit{gate}(s) where \( s \in \{\text{entry}, \text{exit}\} \)

\begin{align*}
R_{g1}(s): & \text{"vehicles shall not pass when } x\text{\_gate is closed", see Fig. 10.2} \\
R_{g2}(s): & \text{"vehicle\_pass is forbidden}\after ?x\text{\_pass} \text{\_pass}\ \\
R_{g3}(s): & \text{"x\_gate is forbidden and x\_gate\_open is forbidden}\after !x\text{\_close} \\
& \text{\_close is forbidden} \\
(R_{g1}(s), R_{g2}(s)) & \implies R_{g3}(s)
\end{align*}

**payment**

\begin{align*}
R_{p1}: & \text{"user inserts a coin every time a ticket is inserted and only then", Fig. omitted} \\
R_{p2}: & \text{"user may insert a ticket only initially or after an exit ticket has been issued", Fig. omitted} \\
R_{p3}: & \text{"exit ticket is issued after ticket is inserted and payment is made and only then", Fig. omitted} \\
(R_{p1}, R_{p2}) & \implies R_{p3}
\end{align*}

**supervisor**

\begin{align*}
R_{s1}(\text{entry}): & \text{(assumption borrowed from } \text{gate(\text{entry})}) \\
R_{s2}(\text{entry}): & \text{(assumption borrowed from } \text{gate(\text{entry})}) \\
R_{s3}(\text{exit}): & \text{(assumption borrowed from } \text{gate(\text{exit})}) \\
R_{s4}: & \text{initially and after } \text{\_entry } \text{\_entry close } \text{\_entry\_gate open is forbidden} \\
R_{s5}: & \text{\_entry\_gate open must be enabled} \\
R_{s6}: & \text{\_at most one ticket is issued per vehicle entering the parking and tickets can be issued} \\
& \text{only if requested and ticket is issued only if the parking is not full", see Fig 10.9} \\
R_{s7}: & \text{\_when the entry gate is closed, the entry gate may not open unless a ticket has been issued",} \\
& \text{Fig. omitted} \\
R_{s8}: & \text{\_the entry gate must open when a ticket is issued", Fig. omitted} \\
R_{s9}: & \text{\_exit gate must open after an exit ticket is inserted and only then", Fig. omitted} \\
R_{s10}: & \text{\_exit gate closes only after vehicle has exited parking", Fig. omitted} \\
(R_{s1}(\text{entry}), R_{s2}(\text{exit}), R_{s3}(\text{exit})) & \implies (R_{s4}, \ldots, R_{s10})
\end{align*}

**supervisor: unformalized requirements**

\begin{align*}
R_{s1}: & \text{"the ticket inserted in the gate must be physically the same as the one issued by the payment machine"} \\
R_{s2}: & \text{"a vehicle cannot exit without having paid a ticket"} \\
(R_{s1}(\ldots), R_{s2}(\text{exit}), R_{s3}) & \implies R_{s9}
\end{align*}

**Figure 10.7:** Top-level requirements. Assumptions and Guarantees are written in italics and roman, respectively. The last line of each chapter specifies the causal pairs following (10.6). Quoted requirements are written in natural language: the corresponding formalization as a modal interface is omitted. The additional chapter, in blue, collects requirements not supported by our formalization.

become complex, even for modest sets of requirements. So the formal management of requirements and their translation into formal contracts must be tool-assisted.

Finally, the whole top-level contract \( C \) is the conjunction of the contracts representing its aspects, namely: \textit{entry}/\textit{exit\_gate}, \textit{payment}, and \textit{supervisor}, see formula (10.2). Owing to the complexity of \( C_{\text{\_supervisor}} \) shown in Figure 10.10, we do not show the Modal Interface \( C \) formalizing the full document. Nevertheless, the latter was generated and can then be exploited as we develop next. The above specification only covers the functional viewpoint of the system. Other viewpoints might be of interest as well, e.g., regarding
Figure 10.8: Chapter payment of the top-level requirements document translated into a Modal Interface $C_{\text{payment}}$.

Figure 10.9: Modal Interface for $R_{e,3}$

Figure 10.10: Chapter supervisor of the top-level requirements document translated into a Modal Interface $C_{\text{supervisor}}$ for a capacity of two for the parking garage.

timing behavior and energy consumption. They would be developed with the same method and combined to the above contract $C$ using conjunction.
10.3. Sub-contracting

In this section, we apply the technique developed in Section 8.6 for generating an architecture of sub-systems with their associated sets of requirements. Each sub-system can then be submitted for independent development to a different supplier. The next duty of the designer is, thus, to specify an architecture à la SysML shown on Figure 10.3.

Comment 10.3 (Mismatch requirements architecture vs. system architecture). Observe that the considered instance of the parking garage consists of one entry gate, one exit gate, and one payment machine. Compare with the top-level specification of Figure 10.7. The latter comprises a generic gate, a payment machine, and a supervisor, each one with its set of requirements. Its results in the conjunction (10.2) defining the top-level contract as the merge of its different aspects. In contrast, the architecture of Figure 10.3 involves no supervisor. The supervisor is meant to be distributed among the two gates. This mismatch between requirements architecture and system architecture is representative of real situations: it is the purpose of this application case to propose solutions for it.

In Figure 10.11 we show the result of applying, to the architecture of Figure 10.3, the Algorithm 1 developed in Section 8.6, which yields by construction a refinement of the top-level contract \( \mathcal{C} \) by a decomposition into local contracts:

\[
\mathcal{C} \geq \mathcal{C}_{\text{EntryGate}} \otimes \mathcal{C}_{\text{ExitGate}} \otimes \mathcal{C}_{\text{PaymentMachine}} \quad (10.7)
\]

Local contract \( \mathcal{C}_{\text{EntryGate}} \) is the most complex one because it involves counting. For the sake of readability, we have assumed a capacity of two. For this example, small capacities less than 20 could be handled with enumerated methods. For larger capacities, symbolic methods would be required. Remarkably enough, the decomposition (10.7) involves small sub-systems compared to \( \mathcal{C}_{\text{supervisor}} \) (Fig. 10.10) and the global contract \( \mathcal{C} \); the restriction operation is to be acknowledged for this strong reduction in size. The main reasons are that the components have few synchronizations and that the decomposition method revealed the parallelism that is hidden in \( \mathcal{C} \).

Comment 10.4 (Comment 10.2, cont'd). Referring to Figure 10.7, recall that requirement \( R_{a,9} \) must be validated against the designed system, either
Figure 10.11: The three restrictions of the global contract $C$ for the three sub-systems Entry-Gate (top), ExitGate (mid), and PaymentMachine (bottom).

manually, or via observer techniques. How can this piece of manual work be combined with our formal analysis? Call $C_{\text{supervisor_informal}}$ the contract corresponding to the supplementary chapter of Figure 10.7. The total top-level contract is in fact given by $C \land C_{\text{supervisor_informal}}$. To check a design against this contract it is enough to check it against $C$ and against $C_{\text{supervisor_informal}}$. The former check involves a complex contract but is tool supported. In contrast, the latter check must rely on observers but involves a simple contract. The bottom line is that contract based requirement engineering supports the combination of tool based and manual checks well.

10.3.4 Consistency, Compatibility, Correctness, Completeness

Requirements capture and management are important matters for discussion with certification bodies. These bodies would typically assess a number
of quality criteria from, e.g., the following list elaborated by INCOSE [158]: Accuracy, Affordability, Boundedness, Complexity, Completeness, Conciseness, Conformance, Consistency, Correctness, Criticality-Level, Orthogonality, Priority, Risk, Unambiguosness, and Verifiability. In this section we focus on four quality criteria that are considered central by certification authorities and are relevant to contracts, namely: Completeness, Correctness, Consistency, and Compatibility.

**Consistency & Compatibility.** Consistency and compatibility have been formally defined in the meta-theory, see Chapter 4 and Table 4.1 therein. In particular, those formal definitions can be formally checked. Thus, the question arises whether these formal definitions suitably reflect the common sense meaning of these terms.

According to the common meaning, a set of requirements is consistent if it is not self-contradicting. The intent is that there is no point in trying to implement an inconsistent set of requirements, as no such implementation is going to exist. As the existence of implementations is the formal definition of consistency according to the meta-theory, it meets the common sense interpretation of this term.

We illustrate consistency on the top-level specification of Figure 10.7. Referring to this table, let us investigate the consistency of the set of requirements \{R_{g,3}(exit), R_{s,6}, R_{s,7}\} under assumptions \{R_{g,1}(exit), R_{g,2}(exit)\}. The following scenario is may-reachable for this modal interface:

1) exit_ticket_insert; 2) exit_gate_open; 3) exit_ticket_insert

After this scenario, event exit_gate_open has modality must in interface R_{s,6} \land R_{s,7}, whereas it has modality cannot in interface R_{g,3}(exit). Thus, the state reached after this scenario is inconsistent. Two options are possible:

1. Return to the designer the message that the set of requirements is inconsistent, since it possesses inconsistent states. This has the merit of leaving the designer with the entire responsibility of avoiding hidden contradictions in its requirements. On the other hand, this puts a heavy burden on the shoulders of the designer since she must have detailed understanding of its specification. Alternatively, one can
2. Reduce the resulting interface by pruning away the inconsistent states, hoping that not all states become inconsistent. (See Section 8.3 regarding interface reduction.) This is the option followed by interface theories of Chapter 8. We have followed it in this application example. It is supported by our tool MICA [70]. From a practical standpoint, this should come with warnings returned to the designer.

According to the common meaning, an architecture of sub-systems, as characterized by their respective specifications, is compatible if these sub-systems “match together”, in that they can be composed and the resulting system can interact with the environment as expected and use cases can be operated as wished. As explained in Table 4.1 of Chapter 4, this is the formal definition of compatibility in the meta-theory. Again, the formal definition of compatibility meets its common sense interpretation.

**Correctness.** Correctness can only be defined with reference to another specification. In contract based design, correctness is not a known concept. We thus propose to specialize “correctness” as one of the following properties, depending on the design step performed:

- “correctness” specializes as: “is a correct implementation of”;
- “correctness” specializes as: “is a correct environment of”;
- “correctness” specializes as: “refines”.

**Completeness.** Completeness raises a difficulty. Although the term “completeness” speaks for itself, it cannot be formally defined what it means to be complete, for a top-level specification in the form of a set of requirements. The reason is that we lack a reference against which completeness could be checked. Hence, the only way to inspect a top-level specification for completeness is to explore it manually. The best help for doing this is to execute the specification. Thus, specifications must be executable. Fortunately, Modal Interfaces are executable and, this way, undesirable behaviors can be revealed.

We illustrate this on the top-level specification of Figure 10.7, in which we change requirement R_{s,6} to: “exit gate must open after an exit ticket is
inserted” (by omitting “and only then”). Lack of completeness is revealed by simulation. The following scenario can occur:
1) exit_ticket_insert; 2) exit_gate_open; 3) vehicle_exit;
4) exit_gate_close; 5) exit_gate_open
where step 5) conforms the specification, due to prior step 1). This scenario shows that vehicles may exit without having inserted an exit ticket, an unwanted behavior. This reveals that the specification was not tight enough, i.e., incomplete. So far for completeness of the top-level specification.

In contrast, completeness can be formally defined when a reference \( C \) is available. We propose to say that \( C' \) is incomplete with reference to \( C \), if

1. \( C' \) does not refine \( C \), but
2. there exists \( C'' \) refining \( C' \), and such that: \( C'' \) is consistent, compatible, and refines \( C' \).

The rationale for this definition is that \( C' \) is not precise enough but can be made so by adding some more requirements. Note that \( C' \) is incomplete with reference to \( C \) if and only if \( C' \land C \) is consistent and compatible. This criterion is of particular relevance when \( C' = \bigotimes_{i \in I} C_i \) is an architecture of sub-contracts, where \( C_i \) is to be submitted to supplier \( i \) for independent development.

10.4 Discussion

**More flexible processes for requirement capture.** In our example we assumed that the top-level specification was only structured into aspects, with no reference to sub-systems and system architecture. A more realistic situation would be the following. At the top-level, the system architect performs the following:

- She defines one or several system architectures. At this top-level, the system architecture refers to broad sub-systems in limited numbers. For example, for an aircraft, the engines and the landing gear would be considered as sub-systems of this system architecture.

- She considers the different aspects of the system: functions, safety, timing, resources, and more. For the functions, requirements may be stated
directly at the sub-system level. The system-level contract for the functions are then obtained by composing (using $\otimes$) the contracts for each sub-system. In contrast, top-level specifications for the safety must be stated at the level of the overall system — the overall system safety is the concern.

Once top-level requirements are completed and validated, a more detailed architecture is considered, of which the different sub-systems are subcontracted to different suppliers. Being related to the supplier chain, this second architecture may or may not refine the top-level system architecture.

So far our approach does not provide shortcuts for moving from an architecture to a different one unless the latter refines the former. If not, our approach requires forming the contract composition and then decomposing the result according to the second architecture.

**General remarks.** Requirements engineering is considered very difficult. Requirements are typically numerous and very difficult to structure. Requirements concern all aspects of the system: function, performance, energy, reliability/safety. The framework of contracts expressed as Modal Interfaces that we have proposed here improves the situation in a number of aspects. It encompasses a large part of the requirements.\(^4\) It can accommodate different concrete formalisms. In our example, we have blended textual requirements with requirements specified as state machines. Richer formalisms such as Stateflow diagrams can be accommodated in combination with abstract interpretation techniques — this is not developed here. We have shown how to offer formal support to important properties during the process of certification. We have proposed a correct-by-construction approach to the difficult step of moving from the top-level specification in the form of a requirements document, to an architecture of sub-contracts for the suppliers.\(^5\) Not all requirements can be supported by a formal approach — be it contract-based or

---

\(^4\) According to figures that were given to us by industrials, 70-80% of the requirements can be expressed using the style we have developed here. Other requirements typically involve physical characteristics of the system or define the range for some parameters.

\(^5\) The framework of Assume/Guarantee contracts that is used in Chapter 5 does not offer this, because local alphabets are not properly handled. In contrast, Assume/Guarantee contracts are very permissive in how they can be expressed. In particular, dataflow diagrams (Simulink) can be used to express assumptions and guarantees.
10.4. Discussion

different in nature. It is the merit of our contract-based approach to allow for a semi-automatic/semi-manual handling of requirements.
11

Contracts for Timing in Autosar

11.1 Motivation: timing issues in Autosar

Autosar\(^1\) is a world-wide development partnership including almost all players in the automotive domain electronics supply chain. It has been created with the purpose of developing an open industry standard for automotive software architectures. To achieve the technical goals of modularity, scalability, transferability and reusability of functions, Autosar provides a common software component model and a common infrastructure based on standardized interfaces for the different layers. The Autosar project has focused on the objectives of resource independence, standardization of interfaces and portability of code. While these goals are clearly of paramount importance, their achievement may not be sufficient for improving the quality of software systems and ensuring safe system integration.

As for most other embedded systems, the design of car electronics involves functional as well as non-functional properties, assumptions and constraints [153]. In the Autosar design flow, a large part of the effort is devoted to non-functional aspects combining latencies and throughputs, which are critical in computer controlled systems, and the sharing of communication

\(^1\)http://www.autosar.org/
11.2. An example of an Autosar design process

An example of an Autosar design process using scheduling contracts

To illustrate the practical use of the framework of scheduling contracts in Autosar, we consider as an example an excerpt of an exterior light management system for an automobile. We focus on the parts responsible for sensing driver inputs and actuating the rear direction indicator lights and brake lights. With this example we show how typical timing requirements can be expressed in the contract framework and discuss the added value of these contracts for negotiations between OEM and suppliers. Also we reconsider the Autosar methodology by applying development Process 2 described in Chapter 9. All of this allows us to establish contractual specifications along the supply chain on firm bases, even when electronic control units (ECUs) are going to be shared by different suppliers, who independently implement software components for them. Throughout the case-study we assume discrete time slots to have a fixed duration of 1 \( \mu \text{s} \).

Regarding modeling methodology and notations, we will be using both concrete contracts (for the specification of contracts at early steps of the design) and abstract scheduling contracts (when using the contract algebra). We will use the symbols \( C \) and ‘\( C \)' to distinguish between them. The reader is referred to Chapter 9, and particularly Section 9.4, for the material used.

We now explain how the different steps of Process 2 are instantiated in this example.

11.2.1 Step 1 of Process 2

In this step a view of the Virtual Functional Bus System is created. It shows how the system functions interact regardless of any network topology or deployment across multiple ECUs. Step 1 of Process 2 is performed by considering resourceless contracts for this Virtual Functional Bus (VFB) view. All

\(^2\)A case-study from the German SPES2020 project
contracts created in this step have to be \( T \)-closed to prepare for the later steps where resources are added. In these contracts, functions provided by software components, are represented by means of tasks with precedence constraints. Typical constraints imposed by the language of contracts would be latency intervals, synchronization of events and event models. Figure 11.1 shows the VFB architecture of the exterior light management. The system shall control the brake lights in accordance with the driver pressing the brake pedal. The \texttt{TurnLights} component controls the direction indicator lights according to the position of the turn signal lever and the warn lights button. The system shall also implement an emergency stop signal, where warn lights flash in case of severe braking. The graphical notations in Figure 11.1 distinguishes pure data flows (the dashed lines) from control flows (the solid lines), where the latter may also carry data items. For example the task \( \tau_{\text{TL}} \) is only triggered whenever an event occurs on port \( \text{trig}_\text{TL} \). When executing, it will read the data item from port \( \text{emcy} \). In our context, pure data flow wires induce no precedence constraint: reads from the wire occur independently from corresponding writes.

The languages \( L_A \) and \( L_G \) of a (concrete) scheduling contract can be specified by means of the Autosar timing extensions [115]. The concept of \textit{observable events} allows to derive sorts of scheduling components, as well as ports of their tasks. Precedence order \( \rightarrow \) follows from the interconnection of ports of software components.

To avoid heavy textual notations, in the following we denote by the expression \( \delta(X, Y) \) the latency between occurrence of an event at port \( X \) and occurrence of an event at port \( Y \). Further \( S(X, T, J) \) denotes a periodic event
model for occurrences of events at port \( X \), where \( T \) is the period and \( J \) is the jitter. We will also write \( S(x, T) \), if \( J = 0 \). The expression \( \text{exe}T(\tau) \) denotes the execution time interval for task \( \tau \). We use boolean operators to combine such expressions.

In the case-study there is a maximum allowed latency between brake sensing and activating the brake lights. The same applies to flashing the warn lights in case of an emergency brake situation. The resulting top-level contract for the VFB is as follows:

\[
C_{\text{top}} = (A_{\text{top}}, G_{\text{top}}) = \\
\left( S(\text{ext\_pedal}, 20\text{ms}) \land \delta(\text{ext\_pedal}, \text{ext\_brake\_lamp}) \leq 25\text{ms} \land \right) \\
\left( S(\text{trig\_TL}, 20\text{ms}) \land \delta(\text{ext\_pedal}, \text{ext\_rear\_di\_lamp}) \leq 60\text{ms} \right)
\]

\( A_{\text{top}} \) makes explicit an assumption about the frequency of sensor samples of the brake pedal position. These assumptions were not part of the requirements.\(^3\) The guarantee specifies intervals for the latency when new values have been computed and are sent at output ports.

Contracts in saturated form: The calculus of \( A/G \)-contracts developed in Chapter 5 requires that contracts are in saturated form. Thus, if we write \( C_{\text{top}} = (A, G) \), for \( A \) and \( G \) two abstract scheduling components, we must work with its saturated form \( (A, G \cup \neg A) \). For the case where assumptions do not involve resources, we can use Lemma 9.9. To avoid heavy notations, the saturation of contracts will not be explicitly performed in the specifications to follow. But we will have to take care of it when invoking the algebra of contracts.

11.2.2 Step 2 of Process 2

Assuming that components BrakeLights and TurnLights are implemented by two different suppliers, we propose sub-contracts specifying a time budgeting for them. Thereby a clear assignment of responsibilities to the suppliers is achieved. This activity is Step 2 of Process 2. In our case two subcontracts

\(^3\)It is actually not uncommon that some critical assumptions are implicit in requirements documents, which may, at times, become a problem.
Contracts for Timing in Autosar

are specified:

\[
C_{BL} = \left( S(\text{ext\_pedal}, 20\ms), \delta(\text{ext\_pedal}, \text{ext\_brake\_lamp}) \leq 25\ms \land \delta(\text{ext\_pedal}, \text{emcy}) \leq 5\ms \right)
\]

\[
C_{TL} = \left( S(\text{emcy}, 20\ms, 5\ms) \land S(\text{trig\_TL}, 20\ms), \delta(\text{emcy}, \text{ext\_rear\_di\_lamp}) \leq 50\ms \right)
\]

These contracts are still resourceless. To ensure that \(C_{BL}\) and \(C_{TL}\) are correct with respect to the top-level contract, we must prove the refinement

\[
C_{BL} \otimes C_{TL} \preceq C_{top} \quad (11.1)
\]

Observe that (11.1) involves abstract contracts. In the sequel, we write \(C_{BL} = (A_{BL}, G_{BL})\) and similarly for other abstract contracts. For \(C = (A, G)\) a contract, we set \(\overline{G} \overset{\text{def}}{=} G \cup \lnot A\) (interpreted as: \(G\) in the context of \(A\)), so that the saturated form for this contract is \((A, \overline{G})\). Using these notations and Lemma 9.12, to prove (11.1) it is enough to prove the following:

\[
A_{top} \cap \overline{G}_{BL} \cap \overline{G}_{TL} \subseteq G_{top} \quad (11.2)
\]

\[
A_{top} \subseteq (A_{BL} \cap A_{TL}) \cup \lnot (G_{BL} \cap \overline{G}_{TL}) \quad (11.3)
\]

So far none of the above contracts refer to resources, hence Lemma 9.9 applies. Returning to the mapping from concrete to abstract scheduling components developed in Definition 9.5, we see that, if no resource is involved, the mapping \(M \mapsto [M]^A\) is entirely determined by

1. the precedence order between tasks (or ports), and
2. the language \(L\) specified as part of \(M = (K, L)\).

Since the precedence order on tasks is unchanged when moving from \(C_{top}\) to \(C_{BL} \times C_{TL}\), it is enough to reason using the expressions defining the languages of their assumption and guarantee. We do this without further notice in the sequel. Denote by \(L_{A,TL}^1\) the first statement of \(A_{TL}\) in \(C_{TL}\). We first observe that

\[
A_{top} \cap \overline{G}_{BL} \cap \overline{G}_{TL} = A_{top} \cap G_{BL} \cap (G_{TL} \cup \lnot L_{A,TL}^1).
\]

Reasoning on latencies, jitter, and periods, allows to infer that \(L_{A,TL}^1\) is discharged by \(A_{top} \cap G_{BL}\):

\[
A_{top} \cap G_{BL} \subseteq L_{A,TL}^1. \quad (11.4)
\]
11.2. An example of an Autosar design process

More precisely, (11.4) follows from the following implication:

\[ S(\text{ext\_pedal}, 20\text{ms}) \land \delta(\text{ext\_pedal}, \text{emcy}) \leq 5\text{ms} \implies S(\text{emcy}, 20\text{ms}, 5\text{ms}) \]

Thus, \( A_{\text{top}} \cap \overline{G}_{BL} \cap \overline{G}_{TL} = A_{\text{top}} \cap G_{BL} \cap G_{TL} \) follows. Again, reasoning on latencies, jitter, and periods, allows to prove that \( A_{\text{top}} \cap G_{BL} \cap G_{TL} \subseteq G_{top} \) holds, which shows (11.2). Focus next on (11.3). We have \( A_{BL} \cap A_{TL} = A_{top} \cap L_{A,TL}^{1} \).

Hence,

\[
(A_{BL} \cap A_{TL}) \cup \neg(G_{BL} \cap G_{TL}) = (A_{top} \cap L_{A,TL}^{1}) \cup \neg(G_{BL} \cap G_{TL}) \supseteq (A_{top} \cap L_{A,TL}^{1}) \cup \neg G_{BL}
\]

(by (11.4)) \[ \supseteq (A_{top} \cap L_{A,TL}^{1}) \cup (A_{top} \cap \neg L_{A,TL}^{1}) = A_{top} \]

which proves (11.3), and thus also (11.1).

Discussion: The above step of our contract based design methodology leads to the following important observations:

- Verification tools exist that would make the above reasoning automatic. Our purpose, however, in this use case, is to demonstrate that reasoning with contracts does not need to be performed fully automatically. Here, local properties are considered easy for the human and are thus verified manually. In contrast, lifting local properties of contracts to system wide properties is error prone (risk of circular reasoning) and thus relies on the formal algebra of contracts. How assumptions get discharged when composing contracts must be handled carefully, as our analysis of refinement (11.1) showed.

- Manual reasoning can be complemented by the use of observers. Again, it is important that the use of observers when component composition and viewpoint combination both occur is performed correctly. Our development in Section 5.4 provides the formal support for this. For the patterns used here, a framework for checking refinement of contracts using an observer based strategy is described in [134].

So far we did not consider target architecture for computing, that is, we did not consider resources. If tasks get allocated to resources, the need to schedule tasks sharing a same resource may be a cause of failure to meet the top-level contract. In the next section, we further refine our contracts to account for resources.
### 11.2.3 Steps 3 and 4 of Process 2

The next step in the Autosar methodology consists in developing system and sub-system views, where the network topology and deployment of software components to ECUs is defined. Resource aware contracts are now considered. A resource is allocated to each task of the contracts defined for the VFB description. Precedence constraints between tasks are not modified. If tasks of the assumption are pair-wise non-conflicting (see Definition 9.1), this yields a resource aware $T$-closed contract, that port-refines the contract of the VFB.

For the example, the VFB view is further refined and then deployed as the architecture shown in Figure 11.2. The blue dashed boxes denote the previous components BrakeLights and TurnLights from the architecture shown on Figure 11.1. The brown boxes labeled CBE, CAN and RIE, indicate resources allocated to tasks. Deployment is driven by the separation of the sensing and control parts from the actuation part. In addition to allocating resources, execution budgets are specified per task. Mirroring the decomposition of the VFB description, the contract of the system view is the composition of the resource aware sub-contracts $C'_{BL}$ and $C'_{TL}$. The objective here is to define sub-contracts $C^{res}_{BL}$ and $C^{res}_{TL}$, with the architecture shown in Figure 11.2. We know this will not yield a contract refinement. Nevertheless, we will comply with the steps 3 and 4 of the safe Process 2.

![Figure 11.2: Deploying VFB on computing and communication resources.](image)
11.2. An example of an AUTOSAR design process

**Resource budget:** The OEM provides a resource budget for ECUs CBE and RIE and the CAN bus. The resource reservation for the BrakeLights component is modelled by the tasks shown at the top of Figure 11.2. At the bottom of that figure the tasks of the TurnLights components are depicted. Allocation of tasks to resources is indicated by the brown boxes. The OEM specifies that with a period of $20\text{ms}$ a time window of $2–3\text{ms}$ is available for component BrakeLight_SC for sensing and control of the brake light part, and additional $2\text{ms}$ for the calculation of a possible emergency brake situation. This is modeled by the task $\tau_{\text{CBE BL}}^\text{CBE}$ in Figure 11.2, which has an execution time interval of $[4\text{ms}, 5\text{ms}]$ assigned. Further, task $\tau_{\text{BL}}^\text{CBE}$ triggers task $\tau_{\text{BL}}^\text{CAN}$, which has an execution time of $250\mu\text{s}$. The task $\tau_{\text{BL}}^\text{RIE}$ is periodically triggered each $10\text{ms}$ by a timer of the operating system (represented by $\tau_{\text{CLK1}}^{10\text{ms}}$), reading the value of lamp_in and sending control values to the actuator via port ext_brake_lamp. Similar execution time budgets are assigned to the remaining tasks.

**Performing Step 3 of Process 2:** To perform Step 3 of Process 2 we proceed as follows. We first consider the top-level contract $C_{\text{top}} = (A_{\text{top}}, G_{\text{top}})$. This top-level contract is then modified as follows:

**Regarding assumptions:** Tasks relevant to the assumption $A_{\text{top}}$ remain unchanged. We assign them resources that are different from the resources shown on Figure 11.2. We thus comply with Condition 3 of Lemma 9.15.

**Regarding guarantees:** We split tasks $\tau_{\text{BL}}$ and $\tau_{\text{TL}}$ as shown on Figure 11.2, which results in a change of the sort of guarantee $G_{\text{top}}$ that complies with the Conditions 1 of Lemma 9.7. This splitting results in the creation of new ports. In addition, we want this new set of tasks to be scheduled according to fixed priorities. The additional information provided when specifying the resource budget is translated to constraints on the language of the guarantees, which, again, complies with the Conditions 1 of Lemma 9.7.

Performing this results in the resource aware contract shown on Figure 11.3. This contract complies with the conditions of Lemma 9.15: By $\tau' > \tau$ we denote the order regarding their priorities, meaning $\tau'$ would have a higher priority than $\tau$. At this point we have implemented Step 3 of Process 2.
Performing Step 4 of Process 2: We now proceed to Step 4, which consists in decomposing this resource aware global contract into sub-contracts.

To this end we first map concrete contract $C_{\text{res}}^{\text{top}}$ to its abstract form $A_{\text{res}}^{\text{top}}$, which in turn is amenable of Lemma 9.14. Besides assigning concurrent resources to tasks, mapping $A_{\text{res}}^{\text{top}}$ to $A_{\text{res}}^{\text{top}}$ is immediate since resource allocation causes no conflict. Mapping $G_{\text{res}}^{\text{top}}$ to $G_{\text{res}}^{\text{top}}$, however, requires solving a global schedulability analysis. We thus perform this, which results in the following strengthened statements of the guaranteed delays, reflecting the semantics of the scheduling policy:

$$G_{\text{res}}^{\text{top}} = (A_{\text{res}}^{\text{top}}, G_{\text{res}}^{\text{top}}) = \begin{cases} S(\text{ext\_pedal}, 20ms) & \delta(\text{ext\_pedal}, \text{ext\_brake\_lamp}) \leq 25ms \land \\ S(\text{trig\_TL}, 20ms) & \delta(\text{ext\_pedal}, \text{ext\_rear\_di\_lamp}) \leq 60ms \land \\ & \text{exeT}(\tau_{\text{CBE}}^{BL}) \in [4ms, 5ms] \land \\ & \text{exeT}(\tau_{\text{CAN}}^{BL}) \in [250\mu s, 250\mu s] \land \\ & \text{exeT}(\tau_{\text{RIE}}^{BL}) \in [2ms, 2ms] \land \\ & \text{exeT}(\tau_{\text{CBE}}^{TL}) \in [14ms, 14ms] \land \\ & \text{exeT}(\tau_{\text{CAN}}^{TL}) \in [250\mu s, 250\mu s] \land \\ & \text{exeT}(\tau_{\text{RIE}}^{TL}) \in [4ms, 4ms] \land \\ & \tau_{\text{CBE}}^{BL} > \tau_{\text{CBE}}^{TL} \land \tau_{\text{CAN}}^{BL} > \tau_{\text{CAN}}^{TL} \land \tau_{\text{RIE}}^{BL} > \tau_{\text{RIE}}^{TL} \end{cases}$$

Then, as a final step, we proceed to the decomposition by using Lemma 9.14, resulting in the contracts $C_{\text{BL}}^{\text{res}}$ and $C_{\text{TL}}^{\text{res}}$ shown in Figure 11.4. The languages of their assumptions and guarantees are obtained by projecting to their sorts, the language resulting from the scheduling analysis that has been carried out to compute the semantics of $C_{\text{top}}^{\text{res}}$. The asymmetry between local assumptions $A_{\text{BL}}^{\text{res}}$ and $A_{\text{TL}}^{\text{res}}$ is justified by the chosen priorities: Each task $\tau$ belonging to the guarantee of contract $C_{\text{BL}}^{\text{res}}$ has a higher priority than any task $\tau'$ of the
11.2. An example of an Autosar design process

\[ C_{\text{BL}}^{\text{res}} = (A_{\text{BL}}^{\text{res}}, G_{\text{BL}}^{\text{res}}) = \]
\[
\begin{cases}
\delta(\text{ext\_pedal, ext\_brake\_lamp}) \in [4.25\, ms, 15.25\, ms] \wedge \\
\delta(\text{ext\_pedal, emcy}) \in [4\, ms, 5\, ms] \wedge \\
S(\text{ext\_pedal, 20ms}), \\
exT(\tau_{\text{CBE}}^{\text{BL}}) \in [4\, ms, 5\, ms] \wedge \\
exT(\tau_{\text{CAN}}^{\text{BL}}) \in [250\, \mu s, 250\, \mu s] \wedge \\
exT(\tau_{\text{RIE}}^{\text{BL}}) \in [2\, ms, 2\, ms]
\end{cases}
\]

\[ C_{\text{TL}}^{\text{res}} = (A_{\text{TL}}^{\text{res}}, G_{\text{TL}}^{\text{res}}) = \]
\[
\begin{cases}
\delta(\text{emcy, ext\_rear\_di\_lamp}) \in [8.25\, ms, 41.25\, ms] \wedge \\
S(\text{emcy, 20ms, 1ms}) \wedge \\
S(\text{emcy, 20ms, 1ms}) \wedge \\
G_{\text{BL}}^{\text{res}}, \\
exT(\tau_{\text{CBE}}^{\text{TL}}) \in [14\, ms, 14\, ms] \wedge \\
exT(\tau_{\text{CAN}}^{\text{TL}}) \in [250\, \mu s, 250\, \mu s] \wedge \\
exT(\tau_{\text{RIE}}^{\text{TL}}) \in [4\, ms, 4\, ms] \wedge \\
\tau_{\text{CBE}}^{\text{BL}} > \tau_{\text{CBE}}^{\text{TL}} \wedge \tau_{\text{CAN}}^{\text{BL}} > \tau_{\text{CAN}}^{\text{TL}} \wedge \tau_{\text{RIE}}^{\text{BL}} > \tau_{\text{RIE}}^{\text{TL}}
\end{cases}
\]

Figure 11.4: Resource aware sub-contracts implementing Step 4 of Process 2.

guarantee of contract \( C_{\text{TL}}^{\text{res}} \), which is allocated to the same resource as \( \tau \). The asymmetry between local assumptions \( A_{\text{BL}}^{\text{res}} \) and \( A_{\text{TL}}^{\text{res}} \) is justified by the chosen priorities: Each task \( \tau \) belonging to the guarantee of contract \( C_{\text{BL}}^{\text{res}} \) has a higher priority than any task \( \tau' \) of the guarantee of contract \( C_{\text{TL}}^{\text{res}} \), which is allocated to the same resource as \( \tau \).

As we obtained the guarantees of \( C_{\text{BL}}^{\text{res}} \) and \( C_{\text{TL}}^{\text{res}} \) by projections of the result of the scheduling analysis, the premise (9.32) of Lemma 9.14 holds. So it remains to prove the conditions (9.33) of the same Lemma. Denote by \( L_{A,\text{TL}}^{1} \) the first statement of \( A_{\text{TL}}^{\text{res}} \) and by \( L_{A,\text{TL}}^{2} \) the second statement. Then

\[ A_{\text{top}}^{\text{res}} = A_{\text{BL}}^{\text{res}} \cap L_{A,\text{TL}}^{1} \]

holds. Focus now on the conditions of (9.33). For our contracts they translate to

\[ A_{\text{BL}}^{\text{res}} \cup (L_{A,\text{TL}}^{1} \cap L_{A,\text{TL}}^{2} \cap G_{\text{BL}}^{\text{res}}) \supseteq A_{\text{top}}^{\text{res}} \wedge \neg(G_{\text{BL}}^{\text{res}} \wedge G_{\text{TL}}^{\text{res}}) \tag{11.5} \]
\[ \neg(G_{\text{BL}}^{\text{res}} \cup (L_{A,\text{TL}}^{1} \cup L_{A,\text{TL}}^{2} \cup G_{\text{BL}}^{\text{res}})) \supseteq A_{\text{top}}^{\text{res}} \wedge \neg(G_{\text{BL}}^{\text{res}} \wedge G_{\text{TL}}^{\text{res}}) \tag{11.6} \]
\[ \neg(G_{\text{TL}}^{\text{res}} \cup A_{\text{BL}}^{\text{res}}) \supseteq A_{\text{top}}^{\text{res}} \wedge \neg(G_{\text{BL}}^{\text{res}} \wedge G_{\text{TL}}^{\text{res}}) \tag{11.7} \]
\[ A_{\text{BL}}^{\text{res}} \cap (L_{A,\text{TL}}^{1} \cap L_{A,\text{TL}}^{2} \cap G_{\text{BL}}^{\text{res}}) \cup \neg(G_{\text{BL}}^{\text{res}} \wedge G_{\text{TL}}^{\text{res}}) \supseteq A_{\text{top}}^{\text{res}} \tag{11.8} \]
Since $A_{\text{top}}^{\text{res}} \subseteq A_{\text{BL}}^{\text{res}}$ holds, conditions (11.5) and (11.7) are satisfied. For the remaining conditions observe that $A_{\text{BL}}^{\text{res}} \cap G_{\text{BL}}^{\text{res}} \subseteq \mathbb{L}_{A,TL}^2$ holds. This can be proved by reproducing the same reasoning as we did for proving (11.4). With this, (11.6) and (11.8) are implied by

$$\neg G_{\text{BL}}^{\text{res}} \cup A_{\text{top}}^{\text{res}} \supseteq A_{\text{top}}^{\text{res}} \cap \neg(G_{\text{BL}}^{\text{res}} \cap G_{\text{TL}}^{\text{res}})$$

$$A_{\text{top}}^{\text{res}} \cap G_{\text{BL}}^{\text{res}} \cup \neg G_{\text{BL}}^{\text{res}} \cup \neg G_{\text{TL}}^{\text{res}} \supseteq A_{\text{top}}^{\text{res}}$$

and both conditions are satisfied. Whence all conditions of (9.33) are satisfied and consequently $C_{\text{BL}}^{\text{res}} \otimes C_{\text{TL}}^{\text{res}} \leq C_{\text{top}}^{\text{res}}$ holds. So this concludes step 4 of Process 2. That means contracts $C_{\text{BL}}^{\text{res}}$ and $C_{\text{TL}}^{\text{res}}$ can be refined and implemented independently from each other, possibly by different suppliers. System integration remains safe.

### 11.3 Summary and discussion

We have illustrated the use of contracts in the context of AUTOSAR. Our application case was an example of semi-automatic—or semi-manual—use of contracts:

- Manual reasoning was applied for checking undecidable or computationally complex properties of small, local, sub-systems. Manual reasoning is not a formal analysis but it can be reasonably done and cross-checked as part of V&V activities;

- Combining small local proofs into a global system-level proof is error prone if done manually. The algebra of contracts offers formal support for this combination step.

Task scheduling is a resource allocation problem, which, by essence, can only be solved globally. Thus it was certainly not obvious to find a path toward independent development. Supporting the AUTOSAR methodology by a strict contract based approach was not feasible. It is, however, the merit of our approach:

- To properly bound the development steps that do not comply with the rules of contract based design,
To explain how risks at system integration can still be mitigated, with a clear and limited additional discipline regarding resource segregation.

A smooth transition of AUTOSAR toward using contracts seems feasible. Indeed, the current AUTOSAR release is expressive enough to represent real-time contracts. Using the concept of timing chains, both end-to-end deadlines as well as assumed response times for component execution and communication can be expressed. We think that the contract framework developed and used throughout this case-study is particularly valuable for compositional reasoning about scheduling of applications distributed over several resources. This kind of reasoning is currently not supported by AUTOSAR and it could be worthwhile to provide support for it in the autosar timing extensions.

On a more general level about contracts and AUTOSAR, it has to be noted that support for formal specification of different viewpoints is not provided—currently there is no established notion of viewpoints and, hence, no anchor to easily migrate from the current setting (timing and safety extensions) to one supporting multiple viewpoints. For example, viewpoints such as power or security are interesting candidates.

11.4 Bibliographical note

AUTOSAR is a worldwide development partnership of vehicle manufacturers, suppliers and other companies from the electronics, semiconductor and software industry, see [2] for an official introduction to 10 years of AUTOSAR developments. The following sentence taken from this reference properly defines what the scope of AUTOSAR is: “AUTOSAR is not going to standardise the functional internal behaviour of an application, for example algorithms, but the content exchanged between applications.” Regarding our concerns, AUTOSAR provides, as part of its timing extension, standards to express the syntax part of a real-time scheduling problem (referring to Definition 9.1, means to specify sorts) and selected features of the language part (basic timing properties such as periods, latencies etc.). It says nothing regarding how to use these interface data to guarantee safe integration from the timing point of view.

While a comprehensive formal discussion about the execution semantics of AUTOSAR models does not exist, there is some literature addressing parts

\[\text{http://www.autosar.org/}\]
of it. For example, in [157] the authors proposed an abstract formal model to represent AUTOSAR OS programs with timing protection. Their goal is to compute on the one hand schedulability conditions for a set of periodic tasks and on the other hand allowed maximal preemption times by interrupts while keeping the task set schedulable. Whether this development is compositional is not discussed. In [126] the authors developed a formal model for an AUTOSAR multicore RTOS based on the language Promela. The model includes transition systems for tasks, interrupt service routines and also considers critical sections of task, as well as includes the priority ceiling protocol. Besides enabling model checking and detecting for example deadlocks and livelocks, the primary goal is to generate test cases based on the model. While the model has a certain level of detail with regard to the execution semantics of the operating system, its lacks an explicit notion of time.

Regarding the AUTOSAR timing extensions, in [18] a case study is developed and it is discussed how scheduling analysis techniques apply to an AUTOSAR model enriched by concepts from these timing extensions. The work [17] is interesting since the comparison of MARTE and the AUTOSAR timing extensions found in this paper also includes a discussion how specifications are composed. In the AUTOSAR timing extensions a notion of composition is only discussed for end-to-end delays assigned to software components. These delays may be decomposed along with the component architecture. At a later step scheduling analysis determines whether required delays for end-to-end communication are indeed satisfied. Different semantics for end-to-end communication delays are supported. [127] provides a formal foundation for them. MARTE, on the other hand, considers an assumption/guarantee style for the specification of non-functional properties, making it also a suitable language to specify contracts.

It should be noted that these works based on the AUTOSAR timing extensions or on the Timing Augmented Description Language (e.g. [217]) stay on the level of timing properties well established in classical scheduling theory like periods, latencies etc. So composition is discussed based on these properties. However, they do not offer a notion of independent implementability that would also encompass the deployment of multiple components to the same ECUs of a target platform. To support this, task scheduling itself has to be addressed in a compositional manner.
Our conclusion covers different aspects. Where research goes on this topic is of course a concern. Most important, however, is the status of practice and, particularly, the different migration paths toward contract based system design that have been considered. We devote a section to this. Of course, the main contribution of this monograph is the unified vision offered by the meta-theory of contracts. Finally, our two experiments allowed us to push the methodology to its limits and we feel it useful to report the lessons we could draw from this.

12.1 Status of research

Contracts as a philosophy originated both from software engineering and formal verification communities, with the paradigms of Pre/Post-condition or Assume/Guarantee. Contract based design benefits from many advances in research that were not targeted to it: Interface theories were developed by the community of game theory — component and environment being seen as two players in a game. Modalities, aimed to offer more expressive logics, were born at the boundary between logics and formal verification.
It is, however, not until the 2000’s that the concept of contracts presented here as a tool to support system design emerged. In this evolution, various formalisms and theories were borrowed to develop a rigorous framework. This monograph was intended to show the power of a unified theoretical background for contracts, to illustrate the use of contracts in present methodologies and to point out the challenges for its effective use in future applications. The mathematical elegance of the concepts underpinning this area provides confidence in a sustained continuation of the research effort.

12.2 Status of practice

Contracts were popularized for software systems in the 1990’s by Bertrand Meyer, in the community of Object Oriented programming. The methodology advocated for designing software systems with the Eiffel language [201] uses contracts and their refinement. Concurrency is supported by the rely guarantee contracts, see Section 3.1 and Chapter 7. The use of contracts is therefore state-of-practice in Object Oriented programming, where it is part of the toolkit of component based design.

Testing is perhaps the main part of system development. The most difficult part of it consists of the integration tests and system level tests. Design processes and certification authorities pay a lot of attention to system level tests: while backend code generation for safety critical code using certified code generators (e.g., Scade 6) can relieve the designer from unit testing, system level testing remains mandatory. IOCO testing [253] is nowadays a widely used methodology in industry. It consists in submitting test suites to the system, that satisfy certain assumptions on the legal context of use. We have seen that IOCO testing tightly relates to observers for contracts, see [95] for a detailed discussion of this. One can thus say that the contract philosophy is already well deployed in the activity of testing.

What about contracts in systems design? Industrial take up of new methods is only possible if the value proposition outweighs the costs in introducing a new method. Results reported in a sequence of projects triggered by European SPEEDS project¹ [40, 46, 125, 165] (involving Airbus, Bosch, Israeli Aircraft Industries, Magna Steyr and Saab) and a number of follow

¹Contract number 033471
up projects such as DANSE [21], SPES [33], SPES XT [222, Chapters 8 & 10], CESAR [99], MBAT, Crystal have demonstrated migration strategies from existing design processes to a contract based design process along the following stages, with each stage reflecting different trade-offs between migration costs and value proposition. This migration focuses on the very core of contract based design, namely the identification of assumptions regarding the environment or context of use, and the guarantees offered by the system, provided that assumptions are met.

### 12.2.1 Document Assumptions explicitly, in natural language

This first stage enforces a textual and informal characterization of system components deployed separately in distributed systems. As a useful reference, safety engineers in safety analysis are required to explicate failure hypothesis under which different degrees of robustness against failures can be guaranteed. Similarly, in our first stage, the system designer is required to explicate assumptions about the allowed contexts into which the system can be deployed. Such assumptions can (and should) refer to all relevant system aspects, including behaviour, resource requirements, timing, power, packaging, failures, etc. Thinking about the space of allowed system contexts thus forces the designer to explicate upfront what is, too often, only implicitly assumed in the system model, thus leading to late integration errors. Industrial trials and evaluation [1] show that:

1. “it is natural for designers to capture assumptions”, and that
2. this “increases the communication between engineers”.
3. The “additional focus on system interfaces encourages working on requirements in parallel with design”.
4. It “allows for engineers to proceed with design, instead of waiting for guiding information to be available”.
5. It “decreases the distance between requirements and design”.
6. It makes it “easy to locate impact of updates made”.
7. It makes it “easy to record the status of each contract”.
These findings testify different orthogonal value propositions of contracts, in that (2) reduces misunderstandings of requirements between different design teams and helps to gain better understandings of shared interfaces, (3) and (4) support parallelization of design steps and concurrent engineering, thus contributing to an overall objective of reduce system design time, (6) reduces the cost of design iterations, and (7) provides risk monitoring and reduction in overall processes. We note that this substantiates the expected value proposition in an early paper on optimizing design flow through contracts (c.f. [97]). These value propositions have led to reported reductions in integration time ranging from 17% to 59%, and reported reductions in costs for integration and testing ranging from 36% to 54%.

Capturing assumptions in natural language thus provides significant value at low migrations costs. This step can be integrated in the design flow by explicating assumptions as a separate category and linking assumptions to requirements in Excel, IBM Doors, PTC Integrity, dSPACE Synect etc, the latter then allowing to be fully integrated with tools for requirement tracing.

Introducing ontologies into requirement engineering is a well-established approach for improving the quality of requirements by creating company wide or even application domain wide standards for a taxonomy of all relevant design artefacts, such as in civil avionics. This approach is orthogonal to the introduction of contracts. Combining the approaches constitutes a natural migration path to the stage 2 we explain next.

12.2.2 Using formal contracts and automatic observer generation

Using formal contracts assigns unambiguous meaning to contracts. While it is true that doing so reduces the risk of misunderstanding requirements, this potential value proposition is counteracted by the difficulties for engineers to understand formal requirements, let alone formalize requirements themselves. There is thus a significant body of research supporting the transition, from informal (ontology based) contracts, to formalized contract oriented requirements. This is typically achieved by offering libraries of predefined patterns covering typical classes of contracts used in industrial settings. Pattern libraries have been developed for capturing behaviour, safety requirements, real-time requirements, resource usage requirements, power requirements, but as of today there is only limited industrial experience. Still, evidence
of industrial acceptance is demonstrated by the tool-suite from BTC Embedded Systems, notably the BTC Embedded Specifier, also offering support in transitioning from informal requirements to formalized requirements through their Requirement Capturing Wizard. Industrial acceptance has also been demonstrated at ALES, one of the two UTRC research and development hubs in Europe. These tools are currently deployed in actual development processes in some 20 companies world wide.

Once this step has been mastered, the key value proposition from Stage 2 comes from the ability to automatically generate monitors from contracts for SIL, MIL, HIL testing and on-line monitoring of systems, thus conventional simulation and testing processes can be enhanced by detecting requirement violations and assumption violations in existing verification processes, and deriving precise data for requirement coverage. While no data points on reduction of design time and/or integration time are publicly available, Toyota reported that the usage of the tight integration of BTC’s tool suites together with those of dSPACE allowed to deliver their new version of the Prius hybrid car “in one shot”. In a recent product announcement, automatically generated observers from contracts are available in dSPACE test automation desk for HIL testing.

OFFIS has been recently contracted by the VDA to support timing analysis for ADAS/Automated Driving Applications through co-simulation of timing contracts, development models of such systems, and environment models such as those provided by SILAB [59].

The next stage 3 makes deeper use of formal methods of contract based design.

### 12.2.3 Combining formal contracts with formal analysis tools

Whenever requirements are given in a formal language, additional value propositions come from introducing tools into the design flow for formal consistency analysis of requirements, automatic test (or observer) generation,

---

2 [https://www.btc-es.de/en/](https://www.btc-es.de/en/)
Conclusion

and ultimately formal verification. Numerous publications document the underlying analysis techniques and verification methods, and usage of product offerings including such techniques is state of practice for selected classes of safety critical applications. Examples are the Scade Suite Design Verifier\textsuperscript{7} and Simulink Design Verifier [248], which are both based on the Prover Technology proof engine [239, 240], the products of PikeTec\textsuperscript{8} such as TPT, the tool AGATHA [55] developed at CEA/LIST and Uppaal [34] with its derivatives like Uppaal TIGA [77] and ECDAR [102]. Combining such tools with formal contracts allows among others what has been called virtual integration testing, in which the compliance of the known part of the design context of components to their assumptions can be demonstrated with formal analysis tools, and refinement checks can be automated for proving refinement between a systems contract and contracts of its subsystems. Tool support for contract based virtual integration testing is also available from BTC Embedded Systems and ALES [75, 76].

12.3 Advances in contract theories

Contracts have emerged as an important approach to formally support the design of complex cyber-physical systems. A number of contract theories have been proposed to address the various aspects of systems design (e.g., functions, timing, resources). This diversity called for a clarification of what the essence of a contract theory really is.

In this monograph, we proposed a response to this question by developing a meta-theory of contracts, in which components that represent actual designs and contexts of use, and contracts are handled by abstracting away the mathematical syntax for their representation. In this meta-theory, a contract is simply a pair of sets of components: its set of correct implementations and its set of legal environments or contexts of use. Armed only with this abstract definition, we were able to define all fundamental notions related to contracts, namely: refinement, conjunction, parallel composition and quotient. Fundamental properties of contracts such as consistency and compatibility

\textsuperscript{7}http://www.esterel-technologies.com/
\textsuperscript{8}https://www.piketec.com/
followed naturally. All the expected properties for contract theories were established for this meta-theory. Abstractions were defined at the level of the meta-theory and observers were in addition defined for Assume/Guarantee contracts, thus complementing the apparatus with semi-decision procedures to deal with components and contracts involving data.

By construction, this meta-theory needs to be instantiated by defining a mathematical syntax for the class of components and the class of contracts considered. Such an instantiation can be done in different ways, giving rise to different frameworks. We illustrated this for two very different theories, namely Assume/Guarantee contracts and interface theories in which assumptions and guarantees are not explicit. For both cases, the meta-theory helped clarify issues and properties. In particular we were able to establish illuminating links between Assume/Guarantee contracts on the one hand, and Rely/Guarantee reasoning used in object oriented software and Synchronous and Moore Interfaces on the other hand. Assume/Guarantee contracts were refined to address task scheduling.

The development of our meta-theory also helped us identify how contract based design should be equipped with tools. The idea is that a contract manipulation tool should only handle generic operations and properties of contracts, by using the results of the meta-theory and generate corresponding proof obligations. The latter would then be instantiated by calling verification engines dedicated to specialized domains. This philosophy has been followed in part by the MICA tool [70] used in our application case in requirement engineering.

Despite these important advances, the handling of multiple viewpoints is still not fully covered. Whereas we do have the right notion at hand for fusing viewpoints (the conjunction), we lack practical handling of heterogeneity, for example when fusing contracts related to safety or timing, with contracts related to functions. Clearly the problem of heterogeneity already arises for components, prior to considering contracts. The question is: can we benefit from having the meta-theory, which is indeed generic hence domain agnostic? This is still open but seems like an interesting avenue.
12.4 Application of contracts: lessons from our experiments

By considering both assumptions on the context of use and guarantees offered by the system, and by encompassing (functional and non-functional) behaviors, the notion of contract we considered here represents a significant step ahead. We draw some lessons learned from our experiments.

12.4.1 Lessons learned from requirement capture using contracts

The first application case, the parking garage example, exemplified the benefits of using contracts in requirement engineering. We illustrated the use of viewpoints to support modular development of top-level requirements. We showed how responsibilities can be accurately specified, by distinguishing guarantees offered from assumptions on the context of use throughout the entire contract based development process. Our encoding of requirements as Modal Interfaces formalizes the requirements and allows for their execution and exploration. This was the basis for giving formal support to important certification related properties of requirements such as consistency, compatibility, and completeness. The move, from a contract specified at a certain level, to an architecture of sub-contracts at the next level, is generally meant to be performed by hand and then formally verified using refinement checks. We illustrated in our example the possibility to synthesize this refinement step automatically, by only providing a structural specification of the refined architecture, à la SysML. This was made possible thanks to the availability of the MICA tool [70]. Open issues and future work remain regarding the use of multiple viewpoints. Our viewpoints in the parking garage example were homogeneous in nature. Developing and then fusing heterogeneous viewpoints such as function, safety/reliability, quantitative resources, and more, remains largely open. Still, what we could achieve by using contracts for requirement engineering went way beyond our expectations.

12.4.2 Lessons learned from using contracts for timing and schedulability analysis

AUTOSAR advocates a design methodology by which the functions, structured into tasks, are first designed independently of the computing and communication infrastructure, assuming a virtual AUTOSAR run time environment.
We studied the key step by which time budgets are then allocated to tasks and computing resources are assigned. Lack of formal support in AUTOSAR methodology makes this step difficult today, with little guaranteed at system integration phase. We showed the benefit of using contracts for this step. To this end, we developed an adaptation of the Assume/Guarantee contracts that we call scheduling contracts. Our study illustrated the semi-formal/semi-manual use of contracts, which we believe, is key to enable a smooth transition to contract based design. A contract engine (such as the MICA tool presented in Section 10) could be used in combination with both manual reasoning and dedicated formal verification engines as the basis for future development that will make contracts main stream. The scheduling contracts we have developed with the AUTOSAR context in mind, seem delicate using, e.g., considering the recommended Process 2 of Chapter 9. This study, however, reveals that some of the claims made about the AUTOSAR methodology may be optimistic: cautiousness is certainly recommended. Altogether, we feel this to be a useful lesson.

12.4.3 A summary of what contracts can do for the designer

Contracts offer a technical support to legal customer-supplier documents: Concurrent development, both within and across companies, calls for smooth coordination and integration of the different design activities. Properly defining and specifying the different concurrent design tasks is and remains a central difficulty. Obligations must therefore be agreed upon, together with suspensive conditions, seen as legal documents. By clearly establishing responsibilities, our formalization of contracts constitutes the technical counterpart of such legal documents. Contracts are an enabling technology for concurrent development.

Contracts offer support to certification: By providing formal arguments that can assess and guarantee the quality of a design throughout all design phases (including early requirements capture), contracts offer support for certification. By providing sophisticated tools in support of modularity, reuse in certification is made easier.
**Conclusion**

**Contracts improve requirement engineering:** As illustrated in the Parking Garage example, contracts are instrumental in decoupling top-level system architecture from the architecture used for sub-contracting to suppliers. Formal support is critical in choosing alternative solutions and migrating between different architectures with relatively small effort. Of course, contracts are not the only important technology for requirements engineering—traceability is essential and developing domain specific ontologies is also important.

**Contracts comply with formal and semi-formal approaches:** The need for being “completely formal” has hampered for a long time formal verification in many industrial sectors, in which flexibility and intuitive expression in documentation, simulation and testing, were and remain preferred. As the AUTOSAR use case demonstrates, using contracts makes semi-formal design safer. Small analysis steps are within the reach of human reasoning. In contrast, lifting a combination of small local reasoning steps to a system-wide analysis—as required when virtually exploring system integration—is difficult and error prone as it involves the risk of wrong circular reasoning. Relying on contracts provides the formal guidance and support for a correct system integration analysis.

**Contracts can be used in any design process:** Contracts offer an “orthogonal” support that complements existing methodologies. They can be used in any flow as a supporting technology in composing and refining designs.

**12.5 Epilogue**

We have illustrated in this monograph how suppliers can be given sub-contracts that are correct by construction and can be automatically generated from top-level specification. We believe, however, that the semi-assisted/semi-manual use of contracts such as exemplified by our AUTOSAR case study is already a significant help, useful for requirements engineering too. Altogether, a contract engine (such as the MICA tool [70]) can be used in combination with both manual reasoning and dedicated formal verification
12.5. Epilogue

engines — e.g., for targeting the timing viewpoint or the safety viewpoint. This would provide a smooth transition path to contract-based design in practice.
Acknowledgements

This work was funded over the years in part by the European STREP-COMBEST project number 215543, the European projects CESAR of the ARTEMIS Joint Undertaking and the European IP DANSE, the Artist Design Network of Excellence number 214373, the TerraSwarm project funded by the STARnet phase of the Focus Center Research Program (FCRP) administered by the Semiconductor Research Corporation (SRC), the iCyPhy program sponsored by IBM and United Technology Corporation, the VKR Center of Excellence MT-LAB, and the German Innovation Alliance on Embedded Systems SPES2020, and the Austrian Science Fund (FWF) under grants S11402-N23 (RiSE/SHiNE) and Z211-N23 (Wittgenstein Award).
References


References


References


References


References


References


References


References


References


References


References


