Contracts for System Design

Albert Benveniste, Benoit Caillaud, Dejan Nickovic, Roberto Passerone, Jean-Baptiste Raclet, Philipp Reinkemeier, Alberto Sangiovanni-Vincentelli, Werner Damm, Thomas Henzinger, Kim Guldstrand Larsen

Contracts for Systems Design

Albert Benveniste, Benoît Caillaud, Dejan Nickovic
Roberto Passerone, Jean-Baptiste Raclet, Philipp Reinkemeier
Alberto Sangiovanni-Vincentelli, Werner Damm
Tom Henzinger, Kim Larsen

RESEARCH
REPORT
N° 8147
November 2012
Project-Teams S4

ISSN 0249-6399 ISRN INRIA/RR--8147--FR+ENG

Contracts for Systems Design

Albert Benveniste[*], Benoît Caillaud[†], Dejan Nickovic[‡]
Roberto Passerone[§], Jean-Baptiste Raclet[¶], Philipp Reinkemeier[‖]
Alberto Sangiovanni-Vincentelli[**], Werner Damm[††]
Tom Henzinger[‡‡], Kim Larsen

Project-Teams S4

Research Report n° 8147 — November 2012 — 64 pages

This work was funded 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 MARCO FCRP TerraSwarm grant, 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.

[*] INRIA, Rennes, France. corresp. author: Albert.Benveniste@inria.fr
[†] INRIA, Rennes, France
[‡] Austrian Institute of Technology (AIT)
[§] University of Trento, Italy
[¶] IRIT-CNRS, Toulouse, France
[‖] Offis and University of Oldenburg
[**] University of California at Berkeley
[††] Offis and University of Oldenburg
[‡‡] IST Austria, Klosterneuburg
Aalborg University, Danmark

RESEARCH CENTRE
RENNES – BRETAGNE ATLANTIQUE

Campus universitaire de Beaulieu
35042 Rennes Cedex

Abstract: Systems design has become a key challenge and differentiating factor over the last decades for system companies. Aircrafts, trains, cars, plants, distributed telecommunication military or health care systems, and more, involve systems design as a critical step. Complexity has caused system design times and costs to go severely over budget so as to threaten the health of entire industrial sectors. Heuristic methods and standard practices do not seem to scale with complexity so that novel design methods and tools based on a strong theoretical foundation are sorely needed.

Model-based design as well as other methodologies such as layered and compositional design have been used recently but a unified intellectual framework with a complete design flow supported by formal tools is still lacking albeit some attempts at this framework such as Platform-based Design have been successfully deployed.

Recently an ”orthogonal” approach has been proposed that can be applied to all methodologies proposed thus far to provide a rigorous scaffolding for verification, analysis and abstraction/refinement: *contract-based design*. Several results have been obtained in this domain but a unified treatment of the topic that can help in putting contract-based design in perspective is still missing. This paper intends to provide such treatment where contracts are precisely defined and characterized so that they can be used in design methodologies such as the ones mentioned above with no ambiguity. In addition, the paper provides an important link between *interfaces* and contracts to show similarities and correspondences. Examples of the use of contracts in design are provided as well as in depth analysis of existing literature.

Key-words: system design, component based design, contract, interface.

Contrats pour la conception de systèmes

Résumé : Cet article fait le point sur le concept de contrat pour la conception de systèmes. Les contrats que nous proposons portent, non seulement sur des propriétés de typage de leurs interfaces, mais incluent une description abstraite de comportements. Nous proposons une *méta-théorie*, ou, si l'on veut, une théorie générique des contrats, qui permet le développement séparé de sous-systèmes. Nous montrons que cette méta-théorie se spécialise en l'une ou l'autre des théories connues.

Mots-clés : conception des systèmes, composant, contrat, interface.

Contents

I Introduction 6
I-A The Present: System Design 6
I-B The Future: CPS and SoS 6
I-C The Need for a Methodological Effort 6
I-D Contract based design 7
I-E Reader's guide 7

II System Design Challenges 8
II-A Complexity of Systems 8
II-B Complexity of OEM-Supplier Chains 9
II-C Managing Requirements 9
II-D Managing Risks 10
II-E System-wide Optimization 10

III How Challenges have been addressed so far 10
III-A Complexity of Systems and System-wide Optimization 10
III-A1 Layered design 11
III-A2 Component-based design 11
III-A3 The V-model process 11
III-A4 Model-Based Design 12
III-A5 Virtual Integration 12
III-A6 Platform Based Design 13
III-B Complexity of OEM-Supplier Chains: Standardization and Harmonization 13
III-B1 Standardization of design entities 13
III-B2 Harmonization of processes and certification 14
III-C Managing Requirements: Traceability and Multiple Viewpoints 14
III-D Cross-company Shared Risk Management 14
III-E The Need for Contracts 15

IV Contracts: what? why? where? and how? 16
IV-A Contracts 16
IV-A1 Components and their Environment, Contracts 16
IV-B Contract Operators 17
IV-B1 Contract Composition and System Integration 17
IV-B2 Contract Refinement and Independent Development 18
IV-B3 Contract Conjunction and Viewpoint Fusion 18
IV-C Contracts in requirement engineering 19
IV-D Contract Support for Design Methodologies 20
IV-D1 Supporting open systems 20
IV-D2 Managing Requirements and Fusing Viewpoints 20
IV-D3 Design Chain Management, Re-using, and Independent Development 21
IV-D4 Deployment and Mapping 21
IV-E Bibliographical note 23

V A Mathematical Meta-theory of Contracts 24
V-A Components and their composition 24
V-B Contracts 25
V-C Refinement and conjunction 25
V-D Contract composition 26
V-E Quotient 27
V-F Discussion 27
V-G Observers 27
V-H Bibliographical note 28

VI Panorama of concrete theories 29

VII Panorama: Assume/Guarantee contracts 29
VII-A Dataflow A/G contracts 30
VII-B Capturing exceptions 30
VII-C Dealing with variable alphabets 31
VII-D Synchronous A/G contracts 32
VII-E Observers 32
VII-F Discussion 32
VII-G Bibliographical note 32

VIII Panorama: Interface theories 33
VIII-A Components as i/o-automata 33
VIII-B Interface Automata with fixed alphabet 34
VIII-C Modal Interfaces with fixed alphabet 35
VIII-D Modal Interfaces with variable alphabet 37
VIII-E Projecting and Restricting 38
VIII-F Observers 40
VIII-G Bibliographical note 40

IX Panorama: Timed Interface Theories 42
IX-A Components as Event-Clock Automata 42
IX-B Modal Event-Clock Specifications 43
IX-C Bibliographical note 43

X Panorama: Probabilistic Interface Theories 44
X-A Components as Probabilistic Automata 44
X-B Simple Modal Probabilistic Interfaces 45
X-C Bibliographical note 45

XI The Parking Garage, an example in Requirements Engineering 45
XI-A The contract framework 45
XI-B Top level requirements 46
XI-C Formalizing requirements as contracts 46
XI-D Sub-contracting to suppliers 48
XI-E The four "C" 49
XI-E1 Consistency & Compatibility 49
XI-E2 Correctness 50
XI-E3 Completeness 50
XI-F Discussion 50

XII Contracts in the context of AUTOSAR 50
XII-A The AUTOSAR context 50
XII-B The contract framework 51
XII-C Exterior Light Management System . . 51
XII-C1 Function and timing 51
XII-C2 Safety 56
XII-D Integrating Contracts in AUTOSAR . . . 58
XII-E Summary and discussion 59

XIII Conclusion 59
XIII-A What contracts can do for the designer 59
XIII-B Status of research 59
XIII-C Status of practice 59
XIII-D The way forward 59

References 60
*

I. Introduction

A. The Present: System Design

System companies such as automotive, avionics and consumer electronics companies 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.

In 2010, Toyota had to recall 10 Million cars worldwide for reasons that ranged from the infamous sticky accelerator pedals to steering and engine problems. The last recall at the end of August 2010 was for the engine control module. Toyota is not alone in this situation. Most of the automotive makers had one or more major recalls in the recent past (see e.g., http://www.autorecalls.us) involving electronics as well as mechanical parts. Boeing and Airbus Industries had significant delays in the delivery of their latest planes (787 and A380). For the A380, underlying causes were cited as issues in the cabling system, configuration management and design process. In particular, the complexity of the cabin wiring (100,000 wires and 40,300 connectors) was considered a major issue (see http://en.wikipedia.org/wiki/Airbus_A380). The delays caused the departure of both the EADS and Airbus CEOs and of the program manager for the A380 and caused an overall earning shortfall of 4.8 Billion Euros. Boeing originally planned the first flight of the 787 for August 2007 (see http://en.wikipedia.org/wiki/Boeing_787), but after a stream of delay announcements, the actual first flight occurred on December 15, 2009. The delays were caused by a number of unfortunate events and design errors and caused at least a 2.3 Billion USD write-off not counting the claim of Air India of 1 Billion USD damages for delayed delivery and the revenue shortfalls.

These are examples of the devastating effects that design problems may cause. The specific root causes of these problems are complex and relate to a number of issues ranging from design processes and relationships with different departments of the same company and with suppliers to incomplete requirement specification and testing.

B. The Future: CPS and SoS

Many products and services require to take into consideration the interactions of computational and physical processes. Systems where this interaction is tight and needs special care are called *Cyber-Physical Systems* (CPS) [133]. The broad majority of these new applications can be classified as "distributed sense and control systems" that go substantially beyond the "compute" or "communicate" functions, traditionally associated with information technology. These applications have the potential to radically influence how we deal with a broad range of crucial problems facing our society today: for example, national security and safety, including surveillance, energy management and distribution, environment control, efficient and reliable transportation and mobility, and effective and affordable health care. A recurring property of these applications is that they engage all the platform components simultaneously—from data and computing services on the cloud of large-scale servers, data gathering from the sensory swarm, and data access on mobile devices—with significant heterogeneity. These large scale systems composed of subsystems that are themselves systems are now called *Systems of Systems* (SoS) and are heavily investigated.

As the complexity of these systems increases, our inability to rigorously model the interactions between the physical and the cyber sides creates serious vulnerabilities. Systems become unsafe, with disastrous inexplicable failures that could not have been predicted. The challenges in the realization and operation of these CPS and SoS are manifold, and cover a broad range of largely unsolved design and run-time problems. These include: modeling and abstraction, verification, validation and test, reliability and resiliency, multi-scale technology integration and mapping, power and energy, security, diagnostics, and run-time management. Failure to address these challenges in a cohesive and comprehensive way will most certainly delay if not prohibit the widespread adoption of these new technologies.

C. The Need for a Methodological Effort

We believe the most promising means to address the challenges in systems engineering is to employ structured and formal design methodologies that seamlessly and coherently combine the various dimensions of the design space (be it behavior, space or time), that provide the appropriate abstractions to manage the inherent complexity, and that can provide correct-by-construction implementations. The following technology issues must be addressed when developing new approaches to the design of complex systems, CPS and SoS:

- The overall design flows for heterogeneous systems—meant here both in a technical and also an organizational sense—and the associated use of models across traditional boundaries are not well developed and understood. Relationships between different teams inside a same company, or between different stake-holders in the supplier chain, are not well supported by solid technical descriptions for the mutual obligations.
- System requirement capture and analysis is in large part a heuristic process, where the informal text and natural language-based techniques in use today are facing significant challenges. Formal requirement engineering is in its infancy: mathematical models, formal analysis techniques and links to system implementation must be developed.
- Dealing with variability, uncertainty, and life-cycle issues, such as extensibility of a product family, are not well-addressed using available systems engineering methodology and tools.
- Design-space exploration is rarely performed adequately, yielding suboptimal designs where the architecture selection phase does not consider extensibility, re-usability,

and fault tolerance to the extent that is needed to reduce cost, failure rates, and time-to-market.

- The verification and validation of "complex systems," particularly at the system integration phase, where any interactions are complicated and extremely costly to address, is a common need in defense, automotive, and other industries.

The challenge is to address the entire process and not to consider only point solutions of methodology, tools, and models that ease part of the design.

D. Contract based design

It is our goal to offer a new approach to the system design problem that is rigorous and effective in dealing with the problems and challenges described before, and that, at the same time, does not require a radical change in the way industrial designers carry out their task as it cuts across design flows of different type: *contract-based design*.

Contracts in the layman use of the term are established when an OEM must agree with its suppliers on the subsystem or component to be delivered. Contracts involve a legal part binding the different parties and a technical annex that serves as a reference regarding the entity to be delivered by the supplier—in this work we focus on the technical facet of contracts. Contracts can also be used through their technical annex in concurrent engineering, when different teams develop different subsystems or different aspects of a system within a same company.

In this paper, we argue that contracts can be actually used almost everywhere and at nearly all stages of system design, from early requirements capture, to embedded computing infrastructure and detailed design involving circuits and other hardware. Contracts explicitly handle pairs of properties, respectively representing the assumptions on the environment and the guarantees of the system under these assumptions. Intuitively, a contract is a pair

$$\mathcal{C} = (A, G) \text{ of } \{\text{Assumptions, Guarantees}\},$$

characterizing in a formal way 1) under which context the design is assumed to operate, and 2) what its obligations are. Assume/Guarantee reasoning has been known for quite some time, but it has been used mostly as verification mean for the design of software. Our purpose is more ambitious: contract based design with explicit assumptions is a philosophy that should be followed all along the design, with all kinds of models, whenever necessary. Here, the models we mean are rich—not only profiles, types, or taxonomy of data, but also models describing the functions, performances of various kinds (time and energy), and safety. The consideration of rich contracts as above in the industry is still in its infancy. To make contract-based design a technique of choice for system engineers, we must develop:

- Mathematical foundations for contract representation and requirement engineering that enable the design of frameworks and tools;
- A system engineering framework and associated methodologies and tool sets that focus on system requirement modeling, contract specification, and verification at multiple abstraction layers. The framework should address cross-boundary and cross-organizational design activities.

This paper intends to provide a unified treatment of contracts where they are precisely defined and characterized so that they can be used in design with no ambiguity. In addition, the paper provides an important link between *interfaces* and *contracts* to show similarities and correspondences. Examples of the use of contracts in design will be provided as well as in depth analysis of existing literature.

E. Reader's guide

The organization of the paper is explained in Table I.

In Section II a review of the challenges that are faced by the system industry is proposed, followed, in Section III, by an analysis of the design methodologies that have been deployed to cope with them.

Section IV develops a primer on contracts by using a very simple example requiring only elementary mathematical background to be followed. Its purpose is to smoothly introduce the different concepts and operations we need for a contract framework—the restricted case considered is by no means representative of the kind of system we can address using contracts. We then introduce a motivating example representative of requirement engineering. In a third sub-section, "requirements" on a theory of contracts are identified from analyzing what is expected from contract-based design, and in particular the support for: 1) the development of different aspects or viewpoints of a system such as its function, safety, timing, and resource use, by different teams and how to fuse these different aspects, and 2) the possibility for different suppliers to develop independently the different subsystems subcontracted to them. This section concludes with the related bibliography.

Section V is the cornerstone of this paper and it is 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. Concepts, operations, and properties are thus stated in a fairly generic way. Every concrete framework compliant with this meta-theory will inherit these generic properties. The meta-theory focuses on assumptions and guarantees, its formalizes how different aspects or viewpoints of a specification can be integrated, and on which basis independent development by different suppliers can be safely performed.

So far the meta-theory is non effective in that it is not said how component and contracts are effectively represented and manipulated. The subsequent series of sections propose a panorama of major concrete contract frameworks. Section VII develops the Assume/Guarantee contracts. This framework is the most straightforward instance of the meta-theory. It deals with pairs (A, G) of assumptions and guarantees explicitly, A and G being both expressed as properties. This framework

Section	methodological / tutorial	fundamental	technical	application
	Section II challenges			
	Section III addressing the challenges			
	Section IV why contracts, where, and how?			
		Section V contract meta-theory		
			Section VII Assume/Guarantee contracts	
			Section VIII Interface theories	
			Section IX timed	
			Section X probabilistic	
				Section XI requirements engineering
				Section XII AUTOSAR

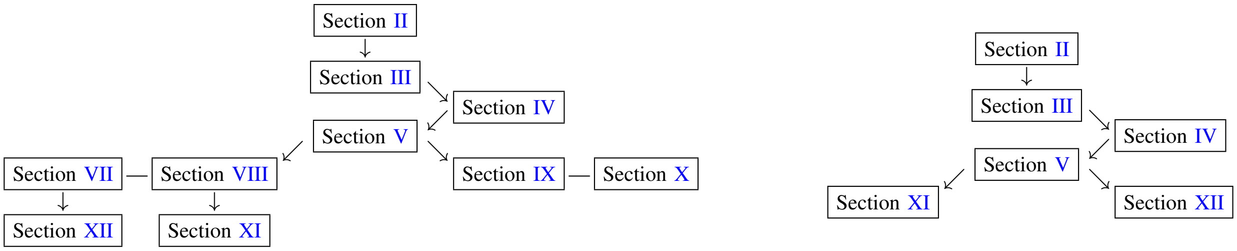

Table I
READER'S GUIDE: CATEGORIZATION OF THE DIFFERENT SECTIONS (TOP), AND DEPENDENCIES (BOTTOM LEFT: IN-DEPTH READING; BOTTOM-RIGHT: QUICK READING)

is flexible in that it allows for different styles of description of such properties—computational efficiency depends on the style adopted. Section VIII develops the Interface theories, in which assumptions and guarantees are specified by means of a single object: the interface. Interface theories turn out to include the most effective frameworks. They can easily be adapted to encompass timing (Section IX) and probabilistic (Section X) characteristics of systems.

In Section XI, a toy example in requirements engineering is developed. This example is simple but rich enough to illustrate the very difficulties in formalizing requirements. It also illustrates well the added value of contract based requirements engineering. First, the different responsibilities (assumptions versus obligation or guarantees) that requirement implicit refer to are made explicit and captured in our contract framework. Second, contracts offer a formal meaning for documents of requirements structured into chapters or viewpoints. Contracts offer formal support for checking important properties such as consistency, compatibility, completeness, and more. Finally, contracts offer significant assistance in the process of deducing, from the top specification, an architecture of specifications for the different sub-systems for independent development. While our application example is a toy one, is it nevertheless too complex for dealing with it by hand. We handle it by using a proof-of-concept tool for contract management.

Finally, Section XII develops a case study in the context of AUTOSAR. AUTOSAR is a recently established standard in the automobile sector. This standard was developed with the objective of allowing for better flexibility in component reuse and OEM/supplier relations. This section illustrates how contracts can be used to break into simple pieces of reasoning the difficult task of correct system integration.

II. System Design Challenges

Many challenges face the system community to deliver products that are reliable and effective. We summarize some of them below.

A. Complexity of Systems

The ever-expanding use of electronic embedded systems to control increasingly many aspects of the real world, and the trend to interconnect more and more such systems (often from different manufacturers) into a global network are creating a challenging scenario for system designers. In this scenario, the three challenges that are taking center stage are as follows.

The Hardware Platform Complexity: Most system companies rely on Components off-the-shelf (COTS) and programmability to implement their applications. For companies who build "physically large" systems such as avionics and automotive companies, the complexity of the hardware platform

Design task	Tasks delayed automotive	Tasks delayed automation	Tasks delayed medical	Tasks causing delay automotive	Tasks causing delay automation	Tasks causing delay medical
System integration test, and verification	63.0%	56.5%	66.7%	42.3%	19.0%	37.5%
System architecture design and specification	29.6%	26.1%	33.3%	38.5%	42.9%	31.3%
Software application and/or middleware development and test	44.4%	30.4%	75.0%	26.9%	31.0%	25.0%
Project management and planning	37.0%	28.3%	16.7%	53.8%	38.1%	37.5%

Table II
DIFFICULTIES RELATED TO SYSTEM INTEGRATION

is reflected in the number of Electronic Control Units and in their interconnections. For a top-of-the-line automobile, the number of processors to manage and interconnect is above 50. The layout of the cables that connect these processing elements with sensors and actuators is a serious concern. Initial production of the Airbus A380 was troubled by delays in part attributed to the 530 km (330 mi) of wiring in each aircraft.

The Embedded Software Complexity: Given the cost and risks associated to developing hardware solutions, system companies are selecting hardware platforms that can be customized by reconfiguration and/or by software programmability. In particular, software is taking the lion's share of the implementation budgets and cost. In cell phones, more than 1 million lines of code is standard today, while in automobiles the estimated number of lines by 2010 is in the order of hundreds of millions and in the Boeing 787 is in the order of 20 million lines. However, as this happens, the complexity explosion of the software component causes serious concerns for the final quality of the products and the productivity of the engineering teams.

The Integration Complexity: A standard technique to deal with complexity consists in decomposing top-down the system into subsystems. This approach, which has been customarily adopted by the semiconductor industry for years, has a limitation as a designer or a group of designers has to fully comprehend the entire system and to partition appropriately its various parts, a difficult task given the enormous complexity of today's systems.

Hence, the future is one of developing systems by composing pieces that all or in part have already been predesigned or designed independently by other design groups or even companies. This has been done routinely in vertical design chains for example in the avionics and automotive sectors, albeit in a heuristic and ad hoc way. The resulting lack of an overall understanding of the interplay of the subsystems and of the difficulties encountered in integrating very complex parts cause system integration to become a nightmare in the system industry, as demonstrated by Table II.[1]

In addition, heterogeneity comes into play. Integration of electronic and mechanical design tools and frameworks will be essential in the near future. Integration of chemical, electronic, and biology tools will be essential in the further future for nano-systems. Data integration and information flow among the companies forming the chain have to be supported. In other words, it is essential that the fundamental steps of system design (functional partitioning, allocation on computational resources, integration, and verification) be supported across the entire design development cycle and across different disciplines.

B. *Complexity of OEM-Supplier Chains*

The source of the above problems is clearly the increase in complexity. It is, however, also the difficulty of the OEMs in managing the integration and maintenance process with subsystems that come from different suppliers who use different design methods, different software architectures, and different hardware platforms.

There are also multiple challenges in defining technical annexes to commercial contracts between OEM and suppliers. Specifications used for procurement should be precise, unambiguous, and complete. However, a recurrent reason for failures causing deep iterations across supply chain boundaries rests in incomplete characterizations of the 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 for interferences through shared resources, and missing boundary conditions. This highlights the need to explicate assumptions on the design context in OEM-supplier commercial contracts. In light of an increased sharing of hardware resources by applications developed by multiple suppliers, a contract-based approach seems indispensable for resolving liability issues and allowing applications with different criticality levels to co-exist (such as ASIL levels[175], [13] in automotive).

C. *Managing Requirements*

We argued that the design chains should connect seamlessly to minimize design errors and time-to-market delays. Yet, the boundaries among companies and between different divisions of the same company are often not as clean as needed and design specs move from one company (or one division) to the

[1] VDC research, Track 3: Embedded Systems Market Statistics Exhibit II-13 from volumes on automotive/industrial automation/medical, 2008

next in non-executable and often unstable and imprecise forms, thus yielding misinterpretations and consequent design errors. In addition, errors are often caught only at the final integration step as the specifications were incomplete and imprecise; further, nonfunctional specifications (e.g., timing, power consumption, size) are difficult to trace. Further, it is common practice to structure system level requirements into several "chapters", "aspects", or "viewpoints". Examples include the functions, safety, timing, and energy viewpoints. Quite often, these different viewpoints are developed by different teams using different skills, frameworks, and tools. Yet they are not unrelated, as we already mentioned. Without a clean approach to handle multiple viewpoints, the only sensible way is to collect these viewpoints into a single requirements document, which is then seen as a flat collection of requirements for subsequent design. Since this is clearly impracticable, the common practice instead is to discard some of the viewpoints in a first stage, e.g., by considering only functions and safety. Designs are then developed based on these only viewpoints. Other viewpoints are subsequently taken into account (timing, energy), thus resulting in late and costly modifications and re-designs.

Requirement engineering is a discipline that aims at improving this situation by paying close attention to the management of the requirement descriptions and traceability support (e.g., using commercial tools such as DOORS[2] in combination with Reqtify[3]) and by inserting whenever possible precise formulation and analysis methods and tools. Research in this area is active but we believe more needs to be done to make this essential step a first class citizen in system design. Indeed, if the specification quality is low, then the entire design process is marred since the very beginning! The overall system product specification is somewhat of an art today since to verify its completeness and its correctness there is little that it can be used to compare with.

D. Managing Risks

System design processes are highly concurrent, distributed, and typically multi-domain, often involving more than one hundred sub-processes. The complexity of the entire design process and of the relationships between players in the supply chain creates the need to elaborate risk sharing and risk management plans because of the potential enormity of the impact that design errors and supplier solidity may have on the economics of a company. Risk management has thus become a requirement of most of public companies, as financial, social and political risks are analyzed and appropriate countermeasures have to be prepared as a requirement from the regulatory bodies. Risk mitigation measures typically cover all phases of design processes, ranging from assuring high quality initial requirements to early assessments of risks in meeting product requirements during the concept phase, to enforcing complete traceability of such requirements with requirements management tools, to managing consistency and synchronization across concurrent sub-processes using PLM tools.[4] If we will be able to change the design process along the lines of more formal approaches, better complexity handling, better requirement engineering, then risks will be substantially lower than they are today.

E. System-wide Optimization

We believe that since the design process is fragmented, product optimization is rarely carried out across more than one company boundary and even then, it is limited due to:

- The lack of appropriate models encompassing both functional and non-functional (Quality of Service) aspects, covering both internal use and export outside the company;
- The time pressure to meet the product deadlines;
- The functional description that is over-constrained by architectural considerations which de facto eliminate potentially interesting implementation alternatives.

If the design process were carried out as in a unique, well-integrated, virtual company including all the players shown above, the overall ecosystem would greatly benefit. The issue here is to allow a reasonably efficient design space exploration by providing a framework where different architectures could be quickly assembled and evaluated at each layer of abstraction corresponding to the design task being considered in the chain.

III. HOW CHALLENGES HAVE BEEN ADDRESSED SO FAR

In this section we present a review and a critical analysis of the design methodologies that have been deployed to cope with the challenges exposed in the previous section.

A. Complexity of Systems and System-wide Optimization

Multiple lines of attack have been developed by research institutions and industry to cope with the exponential growth in systems complexity, starting from the iterative and incremental development several decades ago [123]. Among them, of particular interest to the development of embedded controllers are: Layered design, Component-based design, the V-model process, model-based development, virtual integration and Platform-Based Design (PBD). 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, the V-model process, and model-based development focus on the vertical process while component-based design deals principally with the horizontal process. PBD combines the two aspects in a unified framework and hence subsumes and can be used to integrated

[2] http://www-01.ibm.com/software/awdtools/doors/productline/

[3] http://etc

[4] PLM: Product Lifecycle Management. PLM centric design is used in combination with virtual modeling and digital mockups. PLM acts as a data base of virtual system components. PLM centric design is, for example, deployed at Dassault-Aviation http://www.dassault-aviation.com/en/aviation/innovation/the-digital-company/digital-design/plm-tools.html?L=1

the other methodologies. Contracts are ideal tools to solidify both vertical and horizontal processes providing the theoretical background to support formal methods.

1) Layered design: Layered design copes with complexity by focusing on those aspects of the system pertinent to support the design activities at the corresponding level of abstraction. This approach is particularly powerful if the details of a lower layer of abstraction are encapsulated when the design is carried out at the higher layer. Layered approaches are well understood and standard in many application domains. As an example, consider the AUTOSAR standard.[5] This standard defines several abstraction layers. Moving from "bottom" to "top", the micro-controller abstraction layer encapsulates completely the specifics of underlying micro-controllers, the second layer abstracts from the concrete configuration of the Electronic Control Unit (ECU), the employed communication services and the underlying operating system, whereas the (highest) application layer is not aware of any aspect of possible target architectures, and relies on purely virtual communication concepts in specifying communication between application components. Similar abstraction levels are defined by the ARINC standard in the avionic domains.

The benefits of using layered design are manifold. Using the AUTOSAR layer structure as example, the complete separation of the logical architecture of an application (as represented by a set of components interconnected using the so-called virtual function bus) and target hardware is a key aspect of AUTOSAR, in that it supports decoupling of the number of automotive functions from the number of hardware components. In particular, it is flexible enough to mix components from different applications on one and the same ECU. This illustrates the double role of abstraction layers, in allowing designers to focus completely on the logic of the application and abstracting from the underlying hardware, while at the same time imposing a minimal (or even no) constraint on the design space of possible hardware architectures. In particular, these abstractions allow the application design to be re-used across multiple platforms, varying in number of bus-systems and/or number and class of ECUs. These design layers can, in addition, be used to match the boundaries of either organizational units within a company, or to define interfaces between different organizations in the supply chain. The challenge, then, rests in providing the proper abstractions of lower-level design entities.

2) Component-based design: Whereas layered designs decompose complexity of systems "vertically", component-based approaches reduce complexity "horizontally" whereby designs are obtained by assembling strongly encapsulated design entities called "components" equipped with concise and rigorous interface specifications. Re-use can be maximized by finding the weakest assumptions on the environment sufficient to establish the guarantees on a given component implementation. While these interface specifications are key and relevant for any system, the "quality attribute" of perceiving a subsystem as a component is typically related to two orthogonal criteria, that of "small interfaces", and that of minimally constraining the deployment context, so as to maximize the potential for re-use. "Small interfaces", i.e., interfaces which are both small in terms of number of interface variables or ports, as well as "logically small", in that protocols governing the invocation of component services have compact specifications not requiring deep levels of synchronization, constitute evidence of the success of encapsulation—still, small interfaces must encompass dynamic properties of the system. The second quality attribute is naturally expressible in terms of interface specifications, where re-use can be maximized by finding the weakest assumptions on the environment sufficient to establish the guarantees on a given component implementation.

One challenge, then, for component-based design of embedded systems, is to provide interface specifications that 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. Current component interface models, in contrast, are typically restricted to purely functional characterization of components, and thus cannot capitalize on the benefits of virtual integration testing, as outlined above.

The second challenge is related to product line design, which allows for the joint design of a family of variants of a product. The aim is to balance the contradicting goals of striving for generality versus achieving efficient component implementations. Methods for systematically deriving "quotient" specifications to compensate for "minor" differences between required and offered component guarantees by composing a component with a wrapper component (compensating for such differences as characterized by quotient specifications) exists for restricted classes of component models [155].

3) The V-model process: A widely accepted approach to deal with complexity of systems in the defense and transportation domain is to structure product development processes along variations of the V diagram originally developed for defense applications by the German DoD.[6]

Its characteristic V-shape splits the product development process into a *design* and an *integration* phase. Specifically, following product level requirement analysis, subsequent steps would first evolve a functional architecture supporting product level requirements. Sub-functions are then re-grouped taking into account re-use and product line requirements into a logical architecture, whose modules are typically developed by different subsystem suppliers. The realization of such modules often involves mechanical, hydraulic, electrical, and electronic system design. Subsequent phases would then unfold the detailed design for each of these domains, such as the design of the electronic subsystem involving among others the design of electronic control units. These design phases are paralleled by integration phases along the right-hand part of the V, such as integrating basic- and application software on the ECU hardware to actually construct the electronic control unit, integrating the complete electronic subsystems, integrating the

[5] See http://www.autosar.org/

[6] See e.g. http://www.v-model-xt.de

mechatronic subsystem to build the module, and integrating multiple modules to build the complete product. Not shown, but forming an integral part of V-based development processes are testing activities, where at each integration level test-suites developed during the design phases are used to verify compliance of the integrated entity to their specification.

This presentation is overly simplistic in many ways. The design of electronic components in complex systems such as aircrafts inherently involves multi-site, multi-domain and cross-organizational design teams, reflecting, e.g., a partitioning of the aircraft into different subsystems (such as primary and secondary flight systems, cabin, fuel, and wing), different domains such as the interface of the electronic subsystem to hydraulic and/or mechanical subsystems, control-law design, telecommunications, software design, hardware design, diagnostics, and development-depth separated design activities carried out at the OEM and supplier companies. This partitioning of the design space (along perspectives and abstraction layers) naturally lends itself to a parallelization of design activities, a must in order to achieve timely delivery of the overall product, leading often into the order of hundreds of concurrent design processes. To summarize, while being popular and widely referenced, the V-model process has become a "slogan" hiding the complexity of actual design processes. In reality, large system companies develop their own processes and then struggle for their application, both in-house and across the supplier chain.

4) Model-Based Design: Model-based design (MBD) is today generally accepted as a key enabler to cope with complex system design due to its capabilities to support early requirement validation and virtual system integration. MBD-inspired design languages and tools such as SysML[7] [149] and/or AADL [152] for system level modeling, Catia and Modelica [93] for physical system modeling, Matlab-Simulink [118] for control-law design, and UML[8] [44], [147] Scade [32] and TargetLink for detailed software design, depend on design layer and application class. The state-of-the-art in MBD includes automatic code-generation, simulation coupled with requirement monitoring, co-simulation of heterogeneous models such as UML and Matlab-Simulink, model-based analysis including verification of compliance of requirements and specification models, model-based test-generation, rapid prototyping, and virtual integration testing as further elaborated below.

In MBD today non-functional aspects such as performance, timing, power or safety analysis are typically addressed in dedicated specialized tools using tool-specific models, with the entailed risk of incoherency between the corresponding models, which generally interact. To counteract these risks, meta-models encompassing multiple views of design entities, enabling co-modeling and co-analysis of typically heterogeneous viewpoint specific models have been developed. Examples include the MARTE UML [148] profile for real-time system analysis, the SPEEDS HRC metamodel [156] and the Metropolis semantic meta-model [19], [67], [16], [170]. In Metropolis multiple views are accommodated via the concept of "quantities" that annotate the functional view of a design and can be composed along with subsystems using a suitable algebra. The SPEEDS meta-model building on and extending SysML has been demonstrated to support co-simulation and co-analysis of system models for transportation applications allowing co-assessment of functional, real-time and safety requirements. It forms an integral part of the meta-model-based inter-operability concepts of the CESAR reference technology platform.[9]

Meta-modeling is also at the center of the model driven (software) development (MDD) methodology. MDD is based on the concept of the model-driven architecture (MDA), which consists of a Platform-Independent Model (PIM) of the application plus one or more Platform-Specific Models (PSMs) and sets of interface definitions. MDA tools then support the mapping of the PIM to the PSMs as new technologies become available or implementation decisions change [146]. The Vanderbilt University group [119] has evolved an embedded software design methodology and a set of tools based on MDD. In their approach, models explicitly represent the embedded software and the environment it operates in and capture the requirements and the design of the application, simultaneously, using domain-specific languages (DSL). The generic modeling environment (GME) [119] provides a framework for model transformations enabling easy exchange of models between tools and offers sophisticated ways to support syntactic (but not semantic) heterogeneity. The KerMeta metamodeling workbench [91] is similar in scope.

5) Virtual Integration: Rather than "physically" integrating a system from subsystems at integration stages, model-based design allows systems to be virtually integrated based on the models of their subsystem and the architecture specification of the system. Such virtual integration thus allows detecting potential integration problems up front, in the early design phases. Virtual system integration is often a source of heterogeneous system models, such as when realizing an aircraft function through the combination of mechanical, hydraulic, and electronic systems. Heterogeneous composition of models with different semantics was originally addressed in Ptolemy [88], Metropolis [20], [67], [16], [48], and in the SPEEDS meta-model of heterogeneous rich components [65], [29], [31], albeit with different approaches. Virtual integration involves models of the functions, the computer architecture with its extra-functional characteristics (timing and other resources), and the physical system for control. Some existing frameworks offer significant support for virtual integration: Ptolemy II, Metropolis, and RT-Builder. Developments around Catia and Modelica as well as the new offer SimScape by Simulink provide support for virtual integration of the physical part at an advanced level.

[7] http://www.omg.org/spec/SysML/
[8] http://www.omg.org/spec/UML/
[9] www.cesarproject.eu

6) Platform Based Design: Platform-based design was introduced in the late 1980s to capture a design process that could encompass horizontal (component-based design, virtual integration) and vertical (layered and model-based design) decompositions, and multiple viewpoints and in doing so, support the supply chain as well as multi-layer optimization.

The idea was to introduce a general approach that could be shared across industrial domain boundaries, that would subsume the various definition and design concepts, and that would extend it to provide a framework to reason about design. Indeed, the concepts have been applied to a variety of very different domains: from automotive, to System-on-Chip, from analog circuit design, to building automation to synthetic biology.

The basic tenets of platform-based design are as follows: The design progresses in precisely defined abstraction layers; at each abstraction layer, functionality (what the system is supposed to do) is strictly separated from architecture (how the functionality could be implemented). This aspect is clearly related to layered design and hence it subsumes it.

Each abstraction layer is defined by a design platform. A design platform consists of

- *A set of library components*. This library not only contains computational blocks that carry out the appropriate computation but also communication components that are used to interconnect the computational components.
- *Models of the components that represent a characterization in terms of performance and other non-functional parameters together with the functionality it can support.* Not all elements in the library are pre-existing components. Some may be "place holders" to indicate the flexibility of "customizing" a part of the design that is offered to the designer. In this case the models represent estimates of what can be done since the components are not available and will have to be designed. At times, the characterization is indeed a constraint for the implementation of the component and it is obtained top-down during the refinement process typical of layered designs. This layering of abstractions based on mathematical models is typical of model-based methods and the introduction of non-functional aspects of the design relates to viewpoints.
- *The rules that determine how the components can be assembled and how the the functional and non-functional characteristics can be computed given the ones of the components to form an architecture.*Then, a platform represents a family of designs that satisfies a set of platform-specific constraints [15]. This aspect is related to component-based design enriched with multiple viewpoints.

This concept of platform encapsulates the notion of re-use as a family of solutions that share a set of common features (the elements of the platform). Since we associate the notion of platform to a set of potential solutions to a design problem, we need to capture the process of mapping a functionality (what the system is supposed to do) with the platform elements that will be used to build a platform instance or an "architecture" (how the system does what it is supposed to do). The strict separation between function and architecture as well as the mapping process have been highly leveraged in AUTOSAR. This process is the essential step for refinement and provides a mechanism to proceed towards implementation in a structured way. Designs on each platform are represented by platform-specific design models. A design is obtained by a designer's creating platform instances (architectures) via composing platform components (process that is typical of component-based design), by mapping the functionality onto the components of the architecture and by propagating the mapped design in the design flow onto subsequent abstraction layers that are dealt with in the same way thus presenting the design process as an iterative refinement. This last point dictates how to move across abstraction layers: it is an important part of design space exploration and offers a way of performing optimization across layers. In this respect PBD supports multiple viewpoints in a general way.

B. Complexity of OEM-Supplier Chains: Standardization and Harmonization

So far the main answer to the complexity of OEM-supplier chains has been standardization. Standardization concerns both the design entities and the design processes, particularly through the mechanism of certification.

1) Standardization of design entities: By agreeing on (domain specific) standard representations of design entities, different industrial domains have created their own *lingua franca*, thus enabling a domain wide shared use of design entities based on their standardized representation. Examples of these standards in the automotive sector include the recently approved requirement interchange format standard RIF[10], the AUTOSAR[11] de-facto standard, the OSEK[12] operating system standard, standardized bus-systems such as CAN[13] and Flexray[14], standards for "car2X" communication, and standardized representations of test supported by ASAM[15]. Examples in the aerospace domain include ARINC standards[16] such as the avionics applications standard interface, IMA, RTCA[17] communication standards. In the automation domain, standards for interconnection of automation devices such as Profibus[18] are complemented by standardized design languages for application development such as Structured Text.

As standardization moves from hardware to operating system to applications, and thus crosses multiple design layers, the challenge increases to incorporate all facets of design entities required to optimize the overall product, while at the same time enabling distributed development in complex

[10] http://www.w3.org/2005/rules/wiki/RIF_Working_Group

[11] http://www.autosar.org/

[12] http://www.osek-vdx.org/

[13] http://www.iso.org/iso/search.htm?qt=Controller+Area+Network&searchSubmit=Search&sort=rel&type=simple&published=true

[14] http://www.flexray.com/

[15] http://www.asam.net/

[16] http://www.aeec-amc-fsemc.com/standards/index.html

[17] http://www.rtca.org/

[18] http://www.profibus.com/

supply chains. As an example, to address the different viewpoints required to optimize the overall product, AUTOSAR extended in transitioning from release 3.1 to 4 its capability to capture timing characteristics of design entities, a key prerequisite for assessing alternate deployments with respect to their impact on timing. More generally, the need for overall system optimization calls for the standardization of all non-functional viewpoints of design entities, an objective yet to be achieved in its full generality. in order to allow cross layer optimization. Within the EDA domain, such richness of design interface specifications is industrial practice. Within the systems domain, the rich component model introduced in the Integrated Projects SPEEDS tackles this key objective: in covering multiple design layers (from product requirements to deployment on target architectures) and in providing means of associating multiple viewpoints with each design entity, it is expressive enough to allow for cross-supply chain optimization of product developments. This approach is further elaborated and driven towards standardization in the Artemis flagship project CESAR.

2) Harmonization of processes and certification: Harmonizing or even standardizing key processes (such as development processes and safety processes) provides for a further level of optimization in interactions across the supply chain. As an example, Airbus Directives and Procedures (ADBs) provide requirements for design processes of equipment manufactures. Often, harmonized processes across the supply chain build on agreed maturity gates with incremental acceptance testing to monitor progress of supplier development towards final acceptance, often building on incremental prototypes. Shared use of Product Lifcycle Management (PLM)[19] databases across the supply chain offers further potentials for cross-supply chain optimization of development processes. Also, in domains developing safety related systems, domain specific standards clearly define the responsibilities and duties of companies across the supply chain to demonstrate functional safety, such as in the ISO 26262[20] for the automotive domain, IEC 61508[21] for automation, its derivatives Cenelec EN 50128 and 50126[22] for rail, and Do 178 B[23] for civil avionics.

Yet, the challenge in defining standards rests in balancing the need for stability with the need of not blocking process innovations. As an example, means for compositional construction of safety cases are seen as mandatory to reduce certification costs in the aerospace and rail domains. Similarly, the potential of using formal verification techniques to cope with increasing system complexity is considered in the move from DO 178 B to DO 178 C standards.

C. Managing Requirements: Traceability and Multiple Viewpoints

Depending on application domains, up to 50% of all errors result from imprecise, incomplete, or inconsistent and thus unfeasible requirements. Requirements are inherently ill structured. They must address all aspects of the system and thus cannot be constrained by the *lingua franca* of a particular domain. They are fragmented in the form of large and sometimes huge DOORS files,[24] structured into viewpoints. Two issues have thus been identified by systems industries as essential for requirement management: powerful and flexible traceability tools to relate requirements between them and to link them to tests for validation purposes, and the development of domain specific ontologies as part of the PLM suite. Further, based on an assessment by the system companies processes in the CESAR project[25] and the German Embedded Systems Innovation alliance[26], and building on the finding of the Integrated Project SPEEDS[27], multiple viewpoints have been categorized into *perspectives* and *aspects* to provide a solid requirement and property modeling scaffold. *Perspectives* are viewpoints relevant to architecture modeling, as carried out by different stake-holders during the development process. *Aspects* are viewpoints that are orthogonal to the system structure. Aspects are used to demonstrate compliance of the architecture to end-user concerns. Example of aspects are *safety, cost, maintainability, performance,* and *weight*. The meta-model underlying the CESAR RTP is able to maintain links such as the *satisfy, derive, verify, refine* and allocate relationships between design artifacts of different perspectives and across abstraction layers.

In addition, to cope with the inherently unstructured problem of (in)completeness of requirements, industry has set up domain- and application-class specific methodologies. As particular examples, we mention learning process, such as employed by Airbus to incorporate the knowledge base of external hazards from flight incidents, the Code of Practice proposed by the Prevent Project[28] using guiding questions to assess the completeness of requirements in the concept phase of the development of advanced driver assistance systems. Use-case analysis methods as advocated for UML based development process follow the same objective. All together, requirement engineering misses the support of formal approaches for its structuring and analysis.

D. Cross-company Shared Risk Management

The realization of complex systems calls for design processes that mitigate risks in highly concurrent, distributed, and typically multi-domain engineering processes, often involving more than one hundred sub-processes. Risk mitigation

[19] See [162] and also footnote 4.

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

[21] http://www.iec.ch/functionalsafety/

[22] http://www.cenelec.eu/Cenelec/CENELEC+in+action/Web+Store/Standards/default.htm

[23] http://www.do178site.com/

[24] Typical sub-systems in aeronautics would have a few thousands top-level requirements. An aircraft can have up to several hundred thousands of them.

[25] www.cesarproject.eu

[26] http://spes2020.informatik.tu-muenchen.de

[27] http://www.speeds.eu.com

[28] Albert: http://...

measures typically cover all phases of design processes, ranging from ensuring high quality initial requirements to early assessments of risks in realizability of product requirements during the concept phase, to enforcing complete traceability of such requirements with requirements management tools, to managing consistency and synchronization across concurrent sub-processes using PLM tools. A key challenge rests in balancing risk reduction versus development time and effort: completely eliminating the risks stemming from concurrent engineering essentially requires a complete synchronization along a fine-grained milestone structure, which would kill any development project due to the induced delays. Another key challenge is the "not my fault" syndrome in the OEM-supplier chain when errors call for costly re-design and the issue of penalties arises. Properly assigning responsibilities to risks is still out of reach today.

E. The Need for Contracts

The way system design challenges have been addressed so far leaves huge opportunities for improvements by relying on contract-based design. In this section we briefly summarize, for each existing approach to address the design challenges, how contracts could improve the current situation.

Contribution 1: Addressing Complexity of Systems and System-wide Optimization.
While layered design (Section III-A1) and component based design (Section III-A2) have been critical steps in breaking systems complexity, they do not by themselves provide the ultimate answer. When design is being performed at a considered layer, implicit assumptions regarding other layers (e.g., computing resources) are typically invoked by the designer without having these explicated. Hence, actual properties of these other layers cannot be confronted 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. Contract-based design provides the due discipline, concepts, and techniques to cope with these difficulties.

Model-based development (Section III-A4) has reached significant maturity by covering nearly all aspects of the system (physics, functions, computing resources), albeit not yet in a fully integrated way. To offer a real added value, any newly proposed technology should be rich enough for encompassing all these aspects as well. Contract-based design offers, in large part, this desirable feature.

Virtual integration and virtual modeling (Section III-A5) is a step beyond basic model-based development, by offering an integrated view of all the above different aspects, e.g., physics + functions + performances. Contract-based design supports the fusion of different systems aspects.

Contract-based design is compliant with the V-model process (Section III-A3), just because it does not impose any particular design process. Using contracts only imposes a small set of rules that any process should accommodate, see Section IV-D for a discussion of this.

Finally, Platform-Based Design (PBD, Section III-A6), which is being deployed in some large systems industries such as United Technologies Corporation (UTC), is a systematic and comprehensive methodology that unifies the various design methodologies that have been described in this section. Contracts offer theory and methodological support for both the successive refinement aspect, the composition aspect and the mapping process of PBD allowing formal analysis and synthesis processes.

Contribution 2: Addressing the Complexity of OEM-Supplier Chains.
The problems raised by the complexity of OEM-Supplier Chains (Section II-B) are indeed the core target of contract-based design. By making the explication of implicit assumptions mandatory, contracts help assigning responsibilities to a precise stake holder for each design entity. By supporting independent development of the different sub-systems while guaranteing 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.

Contribution 3: Managing Requirements.
So far the task of getting requirements right and managing them well (Section III-C) has only got support for sorting the complexity out (traceability services and ontologies, which is undoubtely necessary). However, requirements can only be tested on implementations and it is not clear whether proper distinctions are made when performing tests regarding the following: fusing the results of tests associated to different chapters or viewpoints of a requirement document versus fusing the results of tests associated to different sub-systems; testing a requirement under the responsibility of the designer of the considered sub-system versus testing a requirement corresponding to an assumption regarding the context of use of this sub-system—such distinctions should be made, as we shall see. Also, requirements are barely executable and cannot, in general, be simulated. Requirements engineering is the other primary target of contract-based design: the above issues are properly handled by contracts and contracts offer improved support for evidencing the satisfaction of certification constraints.

Contribution 4: Managing Risks.
Risk metrics and risk mitigation measures are the essential tools in managing risks (Section III-D). By offering improved support for sub-contracting and distributing design to different teams, contracts are likely to significantly reduce the "not my fault" syndrome.

Contribution 5: Addressing Certification.
As explained in Section III-B2, the design of critical systems is subject to a number of certification standards. While certification steps are mostly concerned with the processes, not the designs themselves, the recent move from DO 178B to DO 178C for level A critical systems in aeronautics has

shown for the first time the consideration of formal proofs or validations as valid evidences. We believe that the same will eventually hold for contracts. In particular, by providing adequate formal support for completeness, consistency, compatibility, and more, for sets of requirements, contracts would provide a valuable contribution to certification.

IV. Contracts: what? why? where? and how?

As we argued in the previous section, there are two basic principles followed by design methods so far developed: 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 present briefly contracts and the basic operations they support and then me make the point that contracts are ideal tools to solidify both vertical and horizontal processes providing the theoretical background to support formal methods. We conclude the section by providing a (non exhaustive) bibliography on the general concept of contract.

A. Contracts

Here we propose a "primer" on contracts using a simple example as the reference for their use.

1) Components and their Environment, Contracts: We start from a model that consists of a universal set $\mathcal{M}$ 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 exceptional input $y = 0$ by generating an arbitrary output:

$$M_1 : \left\{ \begin{array}{rl} \text{variables:} & \left\{ \begin{array}{rl} \text{inputs:} & x, y \\ \text{outputs:} & z \end{array} \right. \\ \text{types:} & x, y, z \in \mathbb{R} \\ \text{behaviors:} & (y \neq 0 \rightarrow z = x/y) \wedge (y = 0 \rightarrow z = 0) \end{array} \right.$$

A contract *contract*, denoted by the symbol $\mathcal{C}$, is a description of a component with the following characteristic properties:

1) Contracts are intentionally abstract;
2) Contracts distinguish responsibilities of a component from that of its environment

Property 1) of a contract highlights the goal of handling complexity of the systems. Thus, contracts expose enough information about the component, but not more than necessary for the intended purpose. We can see a contract as an underspecified description of a component that can be either very close to the actual component, or specify only a single property of a component's 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 by a number of different components and can operate in a number of different environments. Hence, we define a contract $\mathcal{C}$ at its most abstract level as a pair $\mathcal{C} = (\mathcal{E}_c, \mathcal{M}_c)$ of subsets of components that implement the contract and of subsets of environments in which the contract can operate. We say that a contract $\mathcal{C}$ is *consistent* if $\mathcal{M}_c \neq \emptyset$ and *compatible* if $\mathcal{E}_c \neq \emptyset$.

This definition of contracts and the implementation relation is very general and, as such, it is not effective. In concrete contract- based design theories, a contract needs to have a finite description that does not directly refer to the actual components, and the implementation relation needs to be effectively computable and establish the desired link between a contract and the underlying components that implement it.

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

$$\mathcal{C}_1 : \left\{ \begin{array}{rl} \text{variables:} & \left\{ \begin{array}{rl} \text{inputs:} & x, y \\ \text{outputs:} & z \end{array} \right. \\ \text{types:} & x, y, z \in \mathbb{R} \\ \text{assumptions:} & y \neq 0 \\ \text{guarantees:} & z = x/y \end{array} \right.$$

$\mathcal{C}_1$ defines the set of components having as variables $\{\text{inputs: } x, y; \text{output: } z\}$ of type real, and whose behaviors satisfy the implication

$$\text{"assumptions} \Rightarrow \text{guarantees"}$$

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 $\mathcal{E}_{c_1}$ 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:

$$\mathcal{C}_1' : \left\{ \begin{array}{rl} \text{variables:} & \left\{ \begin{array}{rl} \text{inputs:} & x, y \\ \text{outputs:} & z \end{array} \right. \\ \text{types:} & x, y, z \in \mathbb{R} \\ \text{assumptions:} & \mathrm{T} \\ \text{guarantees:} & z = x/y \end{array} \right.$$

where symbol T 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.

B. Contract Operators

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

1) Contract Composition and System Integration: Intuitively, the composition operator supports component-based design and, in general, horizontal processes. The composition operator, that we denote by the symbol ×, is a partial function on components. The composition is defined with respect to a composability criterion, where two components M and M' are *composable* if their variable types match. Composability is a syntactic property on pairs of components that defines conditions under which the two components can interact. Composition × 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:

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

The 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 they 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 $z = \mathit{abs}(x)$:

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

The component M'_2 is not composable with M_1, because the two components share the same output variable z. Their composition is illegal, 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 involving a more subtle *compatibility* criterion. Two contracts $\mathcal{C}$ and $\mathcal{C}'$ are *compatible* if their variable types match and if there exists an environment in which the two contracts properly interact. The resulting composition $\mathcal{C} \otimes \mathcal{C}'$ should specify, through its assumptions, this set of environments. 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 formalize this. For $\mathcal{C}$ a contract, let $A_\mathcal{C}$ and $G_\mathcal{C}$ be its assumptions and guarantees and define

$$\begin{aligned} G_{\mathcal{C}_1 \otimes \mathcal{C}_2} &= G_{\mathcal{C}_1} \wedge G_{\mathcal{C}_2} \\ A_{\mathcal{C}_1 \otimes \mathcal{C}_2} &= \max \left\{ A \;\middle|\; \begin{array}{l} A \wedge G_{\mathcal{C}_2} \Rightarrow A_{\mathcal{C}_1} \\ \text{and} \\ A \wedge G_{\mathcal{C}_1} \Rightarrow A_{\mathcal{C}_2} \end{array} \right\} \end{aligned} \qquad (1)$$

where "max" refers to the order of predicates by implication; 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 (1) differs from F, the " false" predicate.

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

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

$$\mathcal{C}'_2 : \begin{cases} \text{variables:} & \begin{cases} \text{inputs:} & v \\ \text{outputs:} & y \end{cases} \\ \text{types:} & v, y \in \mathbb{R} \\ \text{assumptions:} & \mathrm{T} \\ \text{guarantees:} & y = -v \end{cases}$$

$\mathcal{C}_2$ specifies components that for any input value u, generate some output x such that $x > u$ and $\mathcal{C}'_2$ specifies components that generate the value of the output variable y as function $y = -v$ of the input v. Observe that both $\mathcal{C}_2$ and $\mathcal{C}'_2$ are consistent. A simple inspection shows that $\mathcal{C}_1$ and $\mathcal{C}_2$ can be composed and their composition yields:

$$\mathcal{C}_1 \otimes \mathcal{C}_2 : \begin{cases} \text{variables:} & \begin{cases} \text{inputs:} & u, y \\ \text{outputs:} & x, z \end{cases} \\ \text{types:} & x, y, u, z \in \mathbb{R} \\ \text{assumptions:} & y \neq 0 \\ \text{guarantees:} & x > u \wedge z = x/y \end{cases}$$

$\mathcal{C}_1$ and $\mathcal{C}'_2$ can also be composed and their composition yields:

$$\mathcal{C}_1 \otimes \mathcal{C}'_2 : \begin{cases} \text{variables:} & \begin{cases} \text{inputs:} & v, x \\ \text{outputs:} & y, z \end{cases} \\ \text{types:} & v, x, y, z \in \mathbb{R} \\ \text{assumptions:} & v \neq 0 \\ \text{guarantees:} & y = -v \wedge z = x/y \end{cases}$$

Both compositions possess a non-empty assumption (observe also that they are both free of exception), reflecting that the two pairs $(\mathcal{C}_1, \mathcal{C}_2)$ and $(\mathcal{C}_1, \mathcal{C}'_2)$ are compatible.

In our example, we require that compositions $\mathcal{C}_1 \otimes (\mathcal{C}_2 \otimes \mathcal{C}_3)$ and $(\mathcal{C}_1 \otimes \mathcal{C}_2) \otimes \mathcal{C}_3$ result in equivalent contracts, as well as compositions $\mathcal{C}_1 \otimes \mathcal{C}_2$ and $\mathcal{C}_2 \otimes \mathcal{C}_1$, thus providing support for incremental system integration.

A *quotient* operation can be defined that is dual to the composition operation. Given a system-wide contract $\mathcal{C}$ and a contract $\mathcal{C}_1$ that specifies pre-existing components and their interactions, the quotient operation $\mathcal{C}/\mathcal{C}_1$ defines the part of the system-wide contract that still needs to be implemented. It formalizes the practice of "patching" a design to make it behave according to another contract.

2) 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 $\mathcal{C}'$ refines contract $\mathcal{C}$, then *any implementation of $\mathcal{C}'$ should 1) implement $\mathcal{C}$ and, 2) be able to operate in any environment for $\mathcal{C}$.* Hence the following definition for refinement pre-order $\preceq$ between contracts: we say that the contract $\mathcal{C}'$ refines the contract $\mathcal{C}$, if $\mathcal{E}_{\mathcal{C}'} \supseteq \mathcal{E}_{\mathcal{C}}$ and $\mathcal{M}_{\mathcal{C}'} \subseteq \mathcal{M}_{\mathcal{C}}$. Since $\preceq$ is a pre-order, refinement is a transitive relation. For our current series of examples, and using previous notations, $\mathcal{C}' \preceq \mathcal{C}$ amounts to requiring that 1/ $A_{\mathcal{C}}$ implies $A_{\mathcal{C}'}$, and 2/ $A_{\mathcal{C}'} \Rightarrow G_{\mathcal{C}'}$ implies $A_{\mathcal{C}} \Rightarrow G_{\mathcal{C}}$.

In Figure 4 we start with the contract $\mathcal{C}_1$ that represents the general system-wide requirements. We decompose these requirements into requirements of three subsystems, resulting in three contracts $\mathcal{C}_{11}$, $\mathcal{C}_{12}$ and $\mathcal{C}_{13}$, with the property that $\mathcal{C}_{11} \otimes \mathcal{C}_{12} \otimes \mathcal{C}_{13} \preceq \mathcal{C}_1$. Now, these three contracts can be handed to different design teams and refined further independently. In particular, $\mathcal{C}_{12}$ is further decomposed into contracts $\mathcal{C}_{121}$ and $\mathcal{C}_{122}$ such that $\mathcal{C}_{121} \otimes \mathcal{C}_{122} \preceq \mathcal{C}_{12}$, and similarly, $\mathcal{C}_{13}$ is further decomposed into contracts $\mathcal{C}_{131}$ and $\mathcal{C}_{132}$ such that $\mathcal{C}_{131} \otimes \mathcal{C}_{132} \preceq \mathcal{C}_{13}$.

For all contracts $\mathcal{C}_1$, $\mathcal{C}_2$, $\mathcal{C}'_1$ and $\mathcal{C}'_2$, if $\mathcal{C}_1$ is compatible with $\mathcal{C}_2$ and $\mathcal{C}'_1 \preceq \mathcal{C}_1$ and $\mathcal{C}'_2 \preceq \mathcal{C}_2$, then $\mathcal{C}'_1$ is compatible with $\mathcal{C}'_2$ and $\mathcal{C}'_1 \otimes \mathcal{C}'_2 \preceq \mathcal{C}_1 \otimes \mathcal{C}_2$.

We now give a concrete example, where we start with very abstract requirements for a component that implements a function $z = x/e^x$. Consider contracts $\mathcal{C}''_1$ and $\mathcal{C}''_2$, that we define as follows:

$$\mathcal{C}''_1 : \left\{ \begin{array}{rl} \text{variables:} & \left\{ \begin{array}{rl} \text{inputs:} & y \\ \text{outputs:} & z \end{array} \right. \\ \text{types:} & y, z \in \mathbb{R} \\ \text{assumptions:} & y \neq 0 \\ \text{guarantees:} & z \in \mathbb{R} \end{array} \right.$$

$$\mathcal{C}''_2 : \left\{ \begin{array}{rl} \text{variables:} & \left\{ \begin{array}{rl} \text{inputs:} & x \\ \text{outputs:} & y \end{array} \right. \\ \text{types:} & x, y \in \mathbb{R} \\ \text{assumptions:} & \mathrm{T} \\ \text{guarantees:} & y > 0 \end{array} \right.$$

The contract $\mathcal{C}''_1$ 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 any real. Note that the contract $\mathcal{C}''_1$ does not declare the nominator input variable x. The contract $\mathcal{C}''_2$ specifies components that have an input variable x and an output variable of type y. The only requirement on the behavior of $\mathcal{C}''_2$ is that y is strictly greater than 0. The composition $\mathcal{C}''_1 \otimes \mathcal{C}''_2$ is well defined. The contract $\mathcal{C}_1$ refines $\mathcal{C}''_1$, since it allows more inputs (the nominator input variable x) and restricts the behavior of the output variable z, by defining its behavior as the division x/y. It follows that $\mathcal{C}_1$ is also compatible with $\mathcal{C}''_2$ and that $\mathcal{C}_1 \otimes \mathcal{C}''_2 \preceq \mathcal{C}''_1 \otimes \mathcal{C}''_2$. Finally, we have that M_1 and M_2 are implementations of their respective contracts. It follows that $M_1 \times M_2$ implements $\mathcal{C}_1 \otimes \mathcal{C}''_2$.

3) Contract Conjunction and Viewpoint Fusion: We now introduce the *conjunction* operator between contracts, denoted by the symbol $\wedge$. Conjunction is complementary to composition:

1) In the early stages of design, the system-level specification consists of a requirements document that is a conjunction of requirements;
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; see Figure 4 for an illustration.
3) Conjunction supports reuse of a component in different parts of a design; see Figure 5 for an illustration.

We state the desired properties of the conjunction operator as follows: Let $\mathcal{C}_1$ and $\mathcal{C}_2$ be two contracts. If $\mathcal{C}_1$ and $\mathcal{C}_2$ are shared refinable, then $\mathcal{C}_1 \wedge \mathcal{C}_2 \preceq \mathcal{C}_1$ and $\mathcal{C}_1 \wedge \mathcal{C}_2 \preceq \mathcal{C}_2$, and for all contracts $\mathcal{C}$, if $\mathcal{C} \preceq \mathcal{C}_1$ and $\mathcal{C} \preceq \mathcal{C}_2$, then $\mathcal{C} \preceq \mathcal{C}_1 \wedge \mathcal{C}_2$.

To illustrate the conjunction operator, we consider a contract $\mathcal{C}_1^T$ that specifies the timing behavior associated with $\mathcal{C}_1$. For this contract, we introduce additional ports that allow us to specify the arrival time of each signal.

$$\mathcal{C}_1^T : \left\{ \begin{array}{rl} \text{variables:} & \left\{ \begin{array}{rl} \text{inputs:} & t_x, t_y \\ \text{outputs:} & t_z \end{array} \right. \\ \text{types:} & t_x, t_y, t_z \in \mathbb{R}_+ \\ \text{assumptions:} & \mathrm{T} \\ \text{guarantees:} & t_z \leq \max(t_x, t_y) + 1 \end{array} \right.$$

The contract $\mathcal{C}_1^T$ is shared refinable with $\mathcal{C}_1$. Their conjunction $\mathcal{C}_1 \wedge \mathcal{C}_1^T$ yields a contract that guarantees, in addition to $\mathcal{C}_1$ itself, a latency with bound 1 (say, in 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 T assumption of the timing contract, and cancels the assumption of $\mathcal{C}_1$. 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 VII. For the moment, we can temporarily fix the problem by introducing y in the interface of the contract, and use it in the assumptions, as in the following contract $\mathcal{C}_2^T$

$$\mathcal{C}_2^T : \left\{ \begin{array}{rl} \text{variables:} & \left\{ \begin{array}{rl} \text{inputs:} & y, t_x, t_y \\ \text{outputs:} & t_z \end{array} \right. \\ \text{types:} & y \in \mathbb{R}; t_x, t_y, t_z \in \mathbb{R}_+ \\ \text{assumptions:} & y \neq 0 \\ \text{guarantees:} & t_z \leq \max(t_x, t_y) + 1 \end{array} \right.$$

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 VII.

C. Contracts in requirement engineering

In this section we introduce a motivating example that is representative of early requirements capture—the specification of a parking garage. Its full development requires technical material that we introduce later and we thus defer it to Section XI. This example illustrates the following features of contract based requirements capture, namely:

- Top-level system specification is by writing a requirements document. Different formalisms may be used for different kinds of requirements. Here we illustrate this by blending textual requirements written in constrained English (possibly obtained using boilerplates) with small sized automata.
- The document itself is structured into *chapters* describing various aspects of the system, such as how gates should behave, how payment should proceed, plus some overall rules.
- Some requirements are under the responsibility of the system under development; they contribute to specifying the *guarantees* that the system offers. Other requirements are not under the responsibility of the system under development; they contribute to defining the *assumptions* regarding the context in which the system should operate as specified.
- Requirements are written in constrained English language or by using automata.

We now begin with the top-level requirements. The system under specification is a parking garage subject to payment by the user. At its most abstract level, the requirements document comprises the different chapters **gate**, **payment**, and **supervisor**, see Table III. The **gate** chapter will collect the generic requirements regarding entry and exit gates. These generic requirements will then be specialized for entry and exit gates, respectively.

gate	
$\mathrm{R}_{\mathrm{g}.1}$:	*"vehicles shall not pass when gate is closed", see Fig. 7*
$\mathrm{R}_{\mathrm{g}.2}$:	*after ?vehicle_pass ?vehicle_pass is forbidden*
$\mathrm{R}_{\mathrm{g}.3}$:	after !gate_open !gate_open is forbidden and after !gate_close !gate_close is forbidden
payment	
supervisor	

Table III
THE TOP-LEVEL SPECIFICATION, WITH CHAPTER **gate** EXPANDED; REQUIREMENTS WRITTEN IN *italics* ARE ASSUMPTIONS UNDER WHICH **gate** SHOULD OPERATE.

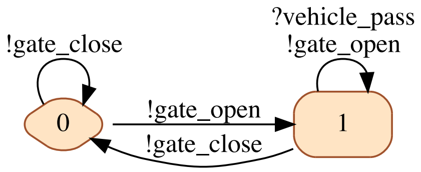

Figure 1. Requirement $\mathrm{R}_{\mathrm{g}.1}$ specified as an i/o-automaton. Prefix "?" indicates an input and prefix "!" indicates an output.

Focus on the "**gate**" chapter. It consists of the three requirements shown on Table III. Requirement $\mathrm{R}_{\mathrm{g}.1}$ is best described by means of an i/o-automaton, shown in Figure 1—we provide an informal textual explanation for it, between quotes. Suppose that some requirement says: "?gate_open never occurs". This is translated by having no mention of ?gate_open in the corresponding i/o-automaton. To express this "negative" fact we must keep track of the fact that ?gate_open belongs to the alphabet of actions of the i/o-automaton. Thus, when performing the translation, the explicit list of inputs and outputs should be explicitly given. To avoid such additional notational burden, we have cheated by omitting this unless necessary. 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 $\mathrm{R}_{\mathrm{g}.1}$ and $\mathrm{R}_{\mathrm{g}.2}$, and 2) we should handle them differently than $\mathrm{R}_{\mathrm{g}.3}$, which is a guarantee offered by the system. Indeed, $\mathrm{R}_{\mathrm{g}.1}$ and $\mathrm{R}_{\mathrm{g}.2}$ are assumptions under which the gate operates as guaranteed. We take the convention that assumptions are written in *italics*.

So far we have specified **gate** as a list of requirements. Requirement $\mathrm{R}_{\mathrm{g}.1}$ specified as an i/o-automaton can be considered formal. Requirements $\mathrm{R}_{\mathrm{g}.2}$ and $\mathrm{R}_{\mathrm{g}.3}$ are formulated in constrained natural language and are ready for subsequent formalization, e.g., as i/o-automata. Are we done? Not yet! We need to give a formal meaning to what it means to have a collection of requirements, and what it means to distinguish assumptions from guarantees. Similarly, we must give a formal meaning to what it means to combine different chapters of a requirements document. Intuitively, all requirements must be met for a chapter to be satisfied, and, similarly, all chapters must be satisfied for the whole specification to be correctly implemented. Thus, we propose to write the top-level specification $\mathcal{C}$ as the following conjunction:

$$\mathcal{C} \quad = \quad \mathcal{C}_{\mathbf{gate}} \wedge \mathcal{C}_{\mathbf{payment}} \wedge \mathcal{C}_{\mathbf{supervisor}}$$

As we said, formally defining what conjunction $\wedge$ is, requires technical material that is developed later in this paper. The following issues arise regarding this top-level specification. First of all, since a conjunction operator is involved in the construction of $\mathcal{C}$, there is a risk of formulating contradicting requirements—this is referred to as the issue of *consistency*. Second, are we sure that the top-level specification $\mathcal{C}$ is *complete*, i.e., precise enough to rule out undesirable implementations? One good way of checking for completeness is to be able to execute or simulate this top-level specification $\mathcal{C}$. We provide answers to all these issues in Section XI, where this example is fully developed.

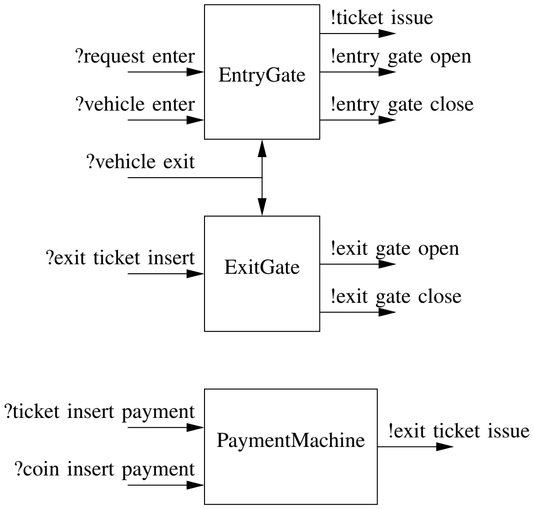

Figure 2. System architecture as specified by the designer.

Having the top-level specification $\mathcal{C}$ at hand, the designer then specifies an architecture “à la SysML”, as shown on Figure 2. Some comments are in order regarding this architecture. The considered instance of a parking garage consists of one entry gate, one exit gate, and one payment machine. Compare with the top-level specification of Table III. The latter comprises a generic gate, a payment machine, and a supervisor, each one with its set of requirements. In contrast, the architecture of Figure 2 involves no supervisor. The supervisor is meant to be distributed among the two gates. The architecture of Figure 2 seems to be very loosely coupled. First, the PaymentMachine seems to be totally independent. In fact, the ticket that is inserted in the exit gate must coincide with the one issued by the PaymentMachine. It turns out that this reflects a missing assumption regarding the environment of the system (namely the user of the parking). Then, the two gates seem to have the shared input “?vehicle exit” as their only interaction. But this shared input is involved in requirement $\mathrm{R}_{\mathrm{s.3}}$, which forbids the entry if the parking is full.

The next step in the design consists in subcontracting the development of each of the three sub-systems of the architecture of Figure 2. This amounts to specifying three subcontracts $\mathcal{C}_{\mathbf{EntryGate}}$, $\mathcal{C}_{\mathbf{ExitGate}}$, and $\mathcal{C}_{\mathbf{PaymentMachine}}$, such that:

$$\mathcal{C}_{\mathbf{EntryGate}} \otimes \mathcal{C}_{\mathbf{ExitGate}} \otimes \mathcal{C}_{\mathbf{PaymentMachine}} \preceq \mathcal{C} \tag{2}$$

The symbol $\preceq$ in (2) means that any implementation of the left hand side is also a valid implementation of the top-level $\mathcal{C}$. 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. Once the three sub-contracts are found, checking that they satisfy (2) requires formalizing the contract composition operator $\otimes$. Furthermore, guessing such sub-contracts is a difficult task. In our development of this example in Section XI, we propose adequate answers to these issues.

D. Contract Support for Design Methodologies

The three operators introduced above fit naturally in any design methodology developed so far. In particular, contract composition and conjunction supports horizontal processes and contract refinement supports vertical processes. We show how this is so by first analyzing the use of contract conjunction for requirement management, then of composition and refinement for design chain management, re-use and independent development to close with conjunction, composition and refinement for deployment and mapping.

1) Supporting open systems: When designing components for reuse, their context of use is not fully known in advance. We thus need a specification of components exposing both the guarantees offered by the component and the assumptions on its possible context of use—its “environment”. This states what *contracts* should be.

2) Managing Requirements and Fusing Viewpoints: Referring to Section III-C, complex systems involve a number of viewpoints (or aspects) that are generally developed by different teams using different skills. As a result, there is a need for fusing these viewpoints in a mathematically sound way. Structuring requirements or specifications is a desirable objective at each step of the design. Finally, we must formalize some key properties that must be evidenced as part of certification processes.

This process is illustrated in Figure 3. In this figure, we show three viewpoints: the behavioral viewpoint where the functions are specified, the timing viewpoint where timing budgets are allocated to the different activities, and the safety viewpoint where fault propagation, effect, and handling, are specified. Typically, different viewpoints are developed by different teams using different frameworks and tools. Development of each viewpoint is performed under assumptions regarding its context of use, including the other viewpoints. To

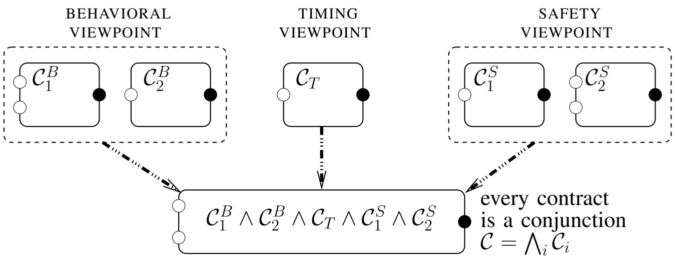

Figure 3. Conjunction of requirements and viewpoints in top-level design

get the full system specification, the different viewpoints must be fused. As the notations of Figure 3 suggest, conjunction is used for fusing viewpoints, thus reflecting that the system under design must satisfy all viewpoints. Similarly, each viewpoint is itself a conjunction of requirements, seen as the "atomic" contracts—all requirements must be met.

3) Design Chain Management, Re-using, and Independent Development: In Figure 4, we show three successive stages of the design. At the top level sits the overall system specification as developed by the OEM. As an example, it can be obtained as the conjunction of several viewpoints as illustrated on Figure 3.

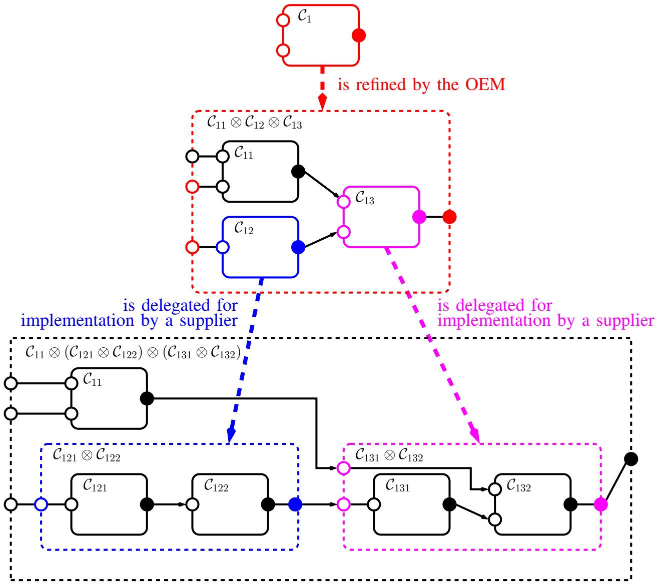
$\mathcal{C}_1$
is refined by the OEM
$\mathcal{C}_{11} \otimes \mathcal{C}_{12} \otimes \mathcal{C}_{13}$
$\mathcal{C}_{11}$
$\mathcal{C}_{13}$
$\mathcal{C}_{12}$
is delegated for
implementation by a supplier
is delegated for
implementation by a supplier
$\mathcal{C}_{11} \otimes (\mathcal{C}_{121} \otimes \mathcal{C}_{122}) \otimes (\mathcal{C}_{131} \otimes \mathcal{C}_{132})$
$\mathcal{C}_{11}$
$\mathcal{C}_{121} \otimes \mathcal{C}_{122}$
$\mathcal{C}_{121}$
$\mathcal{C}_{122}$
$\mathcal{C}_{131} \otimes \mathcal{C}_{132}$
$\mathcal{C}_{131}$
$\mathcal{C}_{132}$

Figure 4. Stepwise refinement

As a first design step, the OEM decomposes its system into an architecture made of three sub-systems for independent development by (possibly different) suppliers. For each of these sub-systems, a contract $\mathcal{C}_{1j}, j = 1, 2, 3$ is developed. A contract composition, denoted by the symbol "$\otimes$",

$$\mathcal{C}_{11} \otimes \mathcal{C}_{12} \otimes \mathcal{C}_{13}$$

mirrors the composition of sub-systems 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 sub-contract $\mathcal{C}_{1j}, j = 1, 2, 3,$ then integrating these sub-systems yields an implementation that satisfies the composed contract $\mathcal{C}_{11} \otimes \mathcal{C}_{12} \otimes \mathcal{C}_{13}$.* (3)

This contract composition must then be qualified against the top-level contract $\mathcal{C}_1$. This qualification must ensure that any development compliant with $\mathcal{C}_{11} \otimes \mathcal{C}_{12} \otimes \mathcal{C}_{13}$ should also comply with $\mathcal{C}_1$. To ensure substitutability in any legal context, compliance concerns both how the system behaves and what its allowed contexts of use are: any legal context for $\mathcal{C}_1$ should be also legal for $\mathcal{C}_{11} \otimes \mathcal{C}_{12} \otimes \mathcal{C}_{13}$ and, under any legal context, the integrated system should behave as specified by $\mathcal{C}_1$. This is formalized as the *refinement* relation, denoted by the symbol $\preceq$:

$$\mathcal{C}_{11} \otimes \mathcal{C}_{12} \otimes \mathcal{C}_{13} \quad \preceq \quad \mathcal{C}_1 \tag{4}$$

Overall, the satisfaction of (4) guarantees the correctness of this first design step performed by the OEM.

Obtaining the three sub-contracts $\mathcal{C}_{11}, \mathcal{C}_{12}$, and $\mathcal{C}_{13}$, 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 (4) holds for the decomposition step or not.
- In passing, the compatibility of the three sub-contracts $\mathcal{C}_{11}, \mathcal{C}_{12}$, and $\mathcal{C}_{13}$, 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 needed information to develop its sub-system in a way that subsequent system integration will be correct.

Each supplier can then proceed with the independent development of the sub-system it is responsible for. For instance, a supplier may reproduce the above procedure.

Alternatively, this supplier can develop some sub-systems by reusing off-the-shelf components. For example, contract $\mathcal{C}_{121}$ would be checked against the interface specification of a pre-defined component M_{121} available from a library, and the following would have to be verified: does component M_{121} satisfy $\mathcal{C}_{121}$? In this context, *shared implementations* are of interest. This is illustrated on Figure 5 where the same off-the-shelf component implements the two referred contracts.

To conclude on this analysis, the two notions of refinement, *denoted by the symbol "$\preceq$", and* composition *of contracts, denoted by the symbol "$\otimes$", are key. Condition (3) ensures independent development holds.*

4) Deployment and Mapping: Here we address a specific but important point related to Contribution 1. At some point in the design of the system, specifications must be realized by using resources. Resources can be pre-defined sub-systems or components, or they can consist of computing resources comprising computing units and communication media (networks, busses, and protocols). This methodology was advocated and

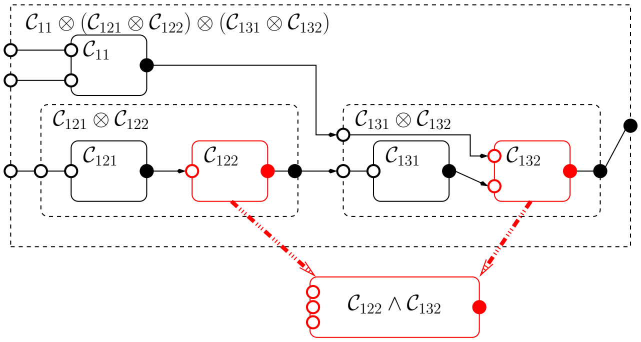
$\mathcal{C}_{11} \otimes (\mathcal{C}_{121} \otimes \mathcal{C}_{122}) \otimes (\mathcal{C}_{131} \otimes \mathcal{C}_{132})$
$\mathcal{C}_{11}$
$\mathcal{C}_{121} \otimes \mathcal{C}_{122}$
$\mathcal{C}_{121}$
$\mathcal{C}_{122}$
$\mathcal{C}_{131} \otimes \mathcal{C}_{132}$
$\mathcal{C}_{131}$
$\mathcal{C}_{132}$
$\mathcal{C}_{122} \wedge \mathcal{C}_{132}$

Figure 5. Conjunction for component reuse

systematically exploited in *Platform Based Design* [171] or PLM-centric design, see footnote 4 in Section II. How such a step can be captured in contract based design is explained next.

When deploying an application over a computing platform, in addition to the functional viewpoint, non-functional viewpoints (safety, timing, energy and possibly other) are of importance as well. The safety viewpoint is concerned with the risk of primary faults and failures, how they could propagate throughout the system, and what the consequences for the resilience of the overall system would be in terms of failure probability. The timing viewpoint collects timing requirements regarding the application (requirements on latencies, throughput, jitter, and schedulability). The effective satisfaction of safety or timing viewpoints by a considered deployment depends on 1) the supporting execution platform and 2) the mapping of the application to this execution platform. We thus need to formalize what "mapping" means. The reader is referred to Figure 6 for the following discussion.

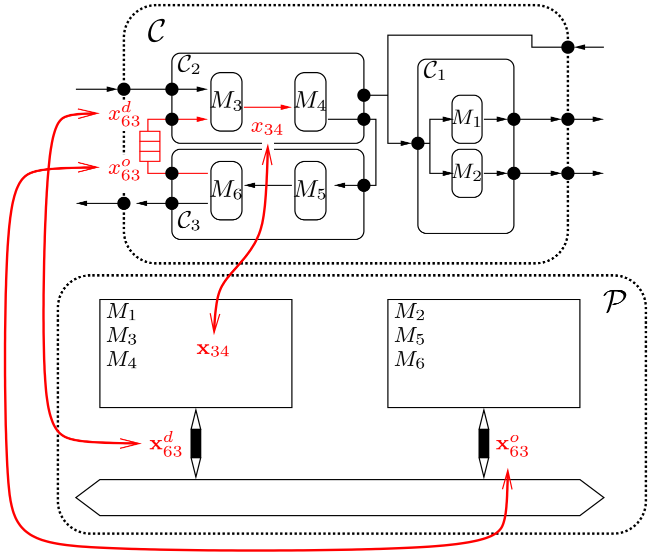

Figure 6. Mapping the application to the execution platform as $\mathcal{C} \wedge \mathcal{P}$.

Suppose one has a virtual model of the execution platform as an architecture composed of several components. Such components would, in the context of Figure 6, consist of a description of the available computing units, a description of the bus protocol and frame structure, and/or a library of RTOS (Real-Time Operating System) services. The resulting components are enhanced with timing information (for the timing viewpoint) and fault propagation information (for the safety viewpoint). Application contracts are attached to the different sub-systems or components as described on the top part of the figure. Similarly, platform contracts are attached to the different sub-systems or components of the execution platform, as described on the bottom part of the figure. Accordingly, we assume

$$\begin{array}{rcl} \mathcal{C} & = & \bigwedge_k \left(\bigotimes_{i\in I_k} \mathcal{C}_{ik}\right) \\ \mathcal{P} & = & \bigotimes_{j\in J} \left(\bigwedge_{\ell\in L_j} \mathcal{P}_{j\ell}\right) \end{array}$$

where:

- The overall platform contract $\mathcal{P}$ decomposes as $\bigotimes_{j\in J} \mathcal{P}_j$ (e.g., the bus and the different computing units in Figure 6) where each $\mathcal{P}_j$ is the conjunction of its different viewpoints (the function: what can be computed; and the associated performance: how much time it takes and how much energy it consumes);
- The overall application contract is the conjunction of its different viewpoints $\mathcal{C}_i$ with their own architectural decomposition.

In Figure 6 we only show a single viewpoint[29] and we assume that distant communication between $\mathcal{C}_2$ and $\mathcal{C}_3$ has already been refined to a communication medium compliant with the communication resources offered by the execution platform.

The mapping of the application over the architecture is modeled by the conjunction $\mathcal{C} \wedge \mathcal{P}$ defined by the following set of *synchronization tuples*,[30] depicted in red in the Figure 6:

- The wire x_{34} linking M_3 to M_4 in the application is synchronized, in the deployment $\mathcal{P}$, with the variable $\mathbf{x}_{34}$ of the left computing unit, where $\mathbf{x}_{34}$ represents the output of the module implementing M_3 and the input of the module implementing M_4. And similarly for the wire linking M_5 to M_6.
- In the deployment, the wire x_{63} linking M_6 to M_3 must traverse the bus. To capture this, we first refine the application architecture by distinguishing the origin x^o_{63} of this wire from its destination x^d_{63}. In the refined application architecture, we assume that x^o_{63} and x^d_{63} are related by some buffer of bounded size. The deployment is then captured by synchronizing x^o_{63} with the output $\mathbf{x}^o_{63}$ of the module implementing M_6 in the right computing unit, and by synchronizing x^d_{63} with the input $\mathbf{x}^d_{63}$ of the module implementing M_3 in the left computing unit.

[29] Here we assume that the different application viewpoints are developed for the whole application and then combined by conjunction—this is just a methodological proposal; we could proceed differently.

[30] In in a synchronization tuple, both occurrences and values of the different elements are unified.

This formalizes the deployment as the conjunction

$$\mathcal{C} \wedge \mathcal{P} = \big[\textstyle\bigwedge_k \big(\bigotimes_{i\in I_k} \mathcal{C}_{ik}\big)\big] \bigwedge \big[\textstyle\bigotimes_{j\in J}\big(\bigwedge_{\ell\in L_j} \mathcal{P}_{j\ell}\big)\big]$$

ensuring that $\mathcal{C} \wedge \mathcal{P}$ refines $\mathcal{C}$, by construction. There is, however, no free lunch. The formula $\mathcal{C} \wedge \mathcal{P}$ expressing deployment involves a conjunction, and is, as such, a possible source of inconsistencies. If this happens, then $\mathcal{C} \wedge \mathcal{P}$ cannot be implemented. Finding an execution platform $\mathcal{P}$ causing no inconsistency in $\mathcal{C} \wedge \mathcal{P}$ requires a good understanding of the application and its needs. The more advanced techniques developed in Section XI provide assistance for this.

Since $\mathcal{C} \wedge \mathcal{P}$ relates application and computing platform—which sit at different layers of the design—we call such contracts *vertical contracts*. Contracts that are not vertical are sometimes called *horizontal*, in that they are meant to relate components sitting at a same level of the design hierarchy—application level, or computing platform level, or ECU (Electronic Computing Unit) level.

Techniques of mapping application to its execution platform that are similar to the above one, are used in the RT-Builder tool [31] and in the Metropolis platform [20], [80].

E. Bibliographical note

Having collected the "requirements" on contract theories, it is now timely to confront these to the previous work referring to or related to the term "contract".

It is difficult to write a comprehensive bibliography on the general aspects of contract based design. The topic is multi faceted and has been addressed by several communities: 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.

Contracts in SW engineering: This part of the bibliographical note was inspired by the report [168]. Design by Contract is a software engineering technique popularized by Bertrand Meyer [139], [140] following earlier ideas from Floyd-Hoare logic [169], [113]. Floyd-Hoare logic assigns meaning to sequential imperative programs in the form of triples of assertions $\{P, C, Q\}$ consisting of a precondition on program states and inputs, a command, and a postcondition on program states and outputs. Meyer's contracts were developed for Object-Oriented programming. They expose the relationships between systems in terms of preconditions and postconditions on operations and invariants on states. A contract on an operation asserts that, given a state and inputs which satisfy the precondition, the operation will terminate in a state and will return a result that satisfy the postcondition and respects any required invariant properties. Contracts contribute to system substitutability. Systems may be replaced by alternative systems or assemblies that offer the same or substitutable functionality with weaker or equivalent preconditions and stronger/equivalent postconditions. With the aim of addressing service oriented architectures, Meyer's contracts were proposed a multiple layering by Beugnard et al. [37]. 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 to prior to service invocation and may also be altered at runtime, thus extending the use of contracts to Systems of Systems [141]. 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 [115] 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.

The concepts of interface and contract were subsequently further developed in the *Model Driven Engineering* (MDE) [120], [172], [131]. 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) [150], [180]. 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. This method, however, does not account for functional and extra-functional properties of interfaces, i.e., for behavior and performance. To account for behavior, the classical approach in MDE consists in enriching components with *methods* that can be invoked from outside, and/or *state machines*. Likewise, attributes on port methods have been used to represent non-functional requirements or provisions of a component [51]. 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 [21], [132], [161]. 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, is unable to cover aspects related to behavior refinement. 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 *Life Sequence Charts* (LSC) in 2000 [64] with its *Play-*

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

Engine implementation [107], David Harel has developed the concept of *behavioral programming* [108], [106], [109], 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.

Our focus—contracts for systems and CPS: The frameworks of contracts developed in the area of Software Engineering have established as useful paradigms for component based software system development. In this paper, we target the wider area of computer controlled systems, more recently referred to as Cyber-Physical systems, where *reactive systems* [104], [110], [100], [136] are encountered, that is systems that continuously interact with some environment, as opposed to *transformational systems* [100], 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 [149] or AADL [152] for system level modeling, Modelica [93] for physical system modeling, Matlab-Simulink [118] for control-law design, and Scade [144], [32] and TargetLink for detailed software design. UML-related standardisation efforts in this area also include the MARTE UML[32] profile for real-time systems. Contract theories for model based development were considered in the community of formal verification. They were initially developed as specification formalisms able to refuse certain inputs from the environment. Dill proposed *asynchronous trace structures* with failure behaviors [81]. 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. Negulescu later generalizes the algebra to Process Spaces which abstract away the specifics of the behaviors, and derives new composition operators [143]. This particular abstraction technique was earlier introduced by Burch with Trace Algebras to construct conservative approximations [47], and later generalized by Passerone and Burch [154] to study generic trace structures with failure behaviors and to formalize the problem of computing the quotient (there called mirror) [153]. This paper 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.

V. A Mathematical Meta-theory of Contracts

In software engineering, meta-models are "models of models", i.e., formal ways of specifying a certain family of models. Similarly, we call here *meta-theory* a way to specify a particular family of theories. In this section we propose a meta-theory of contracts. This meta-theory is summarized in Table IV. It comes as a few primitive concepts, on top of which derived concepts can be built. A number of key properties can be proved about the resulting framework. These properties demonstrate that contracts are a convenient paradigm to support incremental development and independent implementability in system design. The meta-theory will serve as a benchmark for our subsequent review of concrete contract theories.

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. [22] and Chen et al. [59], which follow a different (and complementary) approach. The discussion of this work is deferred to the bibliographical section V-H.

A. Components and their composition

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 abstractions, 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 IV 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* $M_1 \times M_2$ acting on pairs of components. Component composition $\times$ is partially, not to-

[32] www.omgmarte.org

Concept	Definition and generic properties	What depends on the particular theory of contracts
Primitive		
Component	Components are denoted by M; they can be *open* or *closed*	How components are specified
Composability of components	A type property on pairs of components (M_1, M_2)	How this type property is defined
Composition of components	$M_1 \times M_2$ is well defined if and only if M_1 and M_2 are composable; It is required that $\times$ is associative and commutative	The definition of the composition
Environment	An *environment* for component M is a component E such that $E \times M$ is defined and closed	
Derived		
Contract	A *contract* is a pair $\mathcal{C} = (\mathcal{E}_{\mathcal{C}}, \mathcal{M}_{\mathcal{C}})$, where $\mathcal{M}_{\mathcal{C}}$ is a subset of components and $\mathcal{E}_{\mathcal{C}}$ a subset of legal environments	Which family $\mathbf{C}$ of contracts can be expressed, and how they are expressed; unless otherwise specified, quantifying is implicitly over $\mathcal{C} \in \mathbf{C}$
Consistency	$\mathcal{C}$ is *consistent* iff it has at least one component: $\mathcal{M}_{\mathcal{C}} \neq \emptyset$	How consistency is checked
Compatibility	$\mathcal{C}$ is *compatible* iff it has at least one environment: $\mathcal{E}_{\mathcal{C}} \neq \emptyset$	How compatibility is checked
Implementation	$M \models^{\mathrm{M}} \mathcal{C}$ if and only if $M \in \mathcal{M}_{\mathcal{C}}$ $E \models^{\mathrm{E}} \mathcal{C}$ if and only if $E \in \mathcal{E}_{\mathcal{C}}$	How implementation is checked
Refinement	$\mathcal{C}' \preceq \mathcal{C}$ iff $\mathcal{E}_{\mathcal{C}'} \supseteq \mathcal{E}_{\mathcal{C}}$ and $\mathcal{M}_{\mathcal{C}'} \subseteq \mathcal{M}_{\mathcal{C}}$; Property 1 holds	How refinement is checked
GLB and LUB of contracts	$\mathcal{C}_1 \wedge \mathcal{C}_2 =$ Greatest Lower Bound (GLB) for $\preceq$ we assume GLB exist $\mathcal{C}_1 \vee \mathcal{C}_2 =$ Least Upper Bound (LUB) for $\preceq$ we assume LUB exist Property 2 holds	Whether and how GLB and LUB can be expressed and computed
Composition of contracts	$\mathcal{C}_1 \otimes \mathcal{C}_2$ is defined if $\left.\begin{array}{l} M_1 \models^{\mathrm{M}} \mathcal{C}_1 \\ M_2 \models^{\mathrm{M}} \mathcal{C}_2 \end{array}\right\} \Rightarrow (M_1, M_2)$ composable $\mathcal{C}_1 \otimes \mathcal{C}_2 = \bigwedge \left\{ \mathcal{C} \;\middle\vert\; \begin{array}{c} M_1 \models^{\mathrm{M}} \mathcal{C}_1 \text{ and } M_2 \models^{\mathrm{M}} \mathcal{C}_2 \text{ and } E \models^{\mathrm{E}} \mathcal{C} \\ \Downarrow \\ M_1 \times M_2 \models^{\mathrm{M}} \mathcal{C} \text{ and } E \times M_2 \models^{\mathrm{E}} \mathcal{C}_1 \text{ and } E \times M_1 \models^{\mathrm{E}} \mathcal{C}_2 \end{array} \right\}$ Assumption 1 is in force; Properties 3, 4, and 5 hold Say that $\mathcal{C}_1$ and $\mathcal{C}_2$ are *compatible* if so is $\mathcal{C}_1 \otimes \mathcal{C}_2$	How composition is expressed and computed
Quotient	$\mathcal{C}_1 / \mathcal{C}_2 = \bigvee \{ \mathcal{C} \mid \mathcal{C} \otimes \mathcal{C}_2 \preceq \mathcal{C}_1 \}$; Property 6 holds	How quotient is expressed and computed

Table IV
Summary of the meta-theory of contracts. We first list **primitive** concepts and then **derived** concepts introduced by the meta-theory.

tally, defined. Two components that can be composed are called *composable*. Composability of components is meant to be a typing property. In order to guarantee that different composable components may be assembled together in any order, it is required that component composition $\times$ is associative and commutative. Components that exhibit no means for interacting with the outside world are called *closed*; other components are *open*. An *environment* for a component M is another component E composable with M and such that $M \times E$ is closed.

B. Contracts

In our primer of Section IV, we have highlighted the importance of the legal environments associated with contracts, for which an implementation will operate satisfactorily. At the abstract level of the meta-theory, we make this explicit next:

Definition 1: A contract *is a pair $\mathcal{C} = (\mathcal{E}_c, \mathcal{M}_c)$, where:*

- *$\mathcal{M}_c \subseteq \mathcal{M}$ is the set of* implementations *of $\mathcal{C}$, and*
- *$\mathcal{E}_c \subseteq \mathcal{M}$ is the set of* environments *of $\mathcal{C}$.*

For any pair $(E, M) \in \mathcal{E}_c \times \mathcal{M}_c$, E is an environment for M. Hence, $M \times E$ must be well defined and closed. A contract possessing no implementation is called inconsistent. *A contract possessing no environment is called* incompatible. *Write*

$$M \models^{\mathrm{M}} \mathcal{C} \;\; \textit{and} \;\; E \models^{\mathrm{E}} \mathcal{C}$$

to express that $M \in \mathcal{M}_c$ and $E \in \mathcal{E}_c$, respectively.

In concrete theories of contracts, both components and contracts are described in some finite (preferably concise) way. In turn, it is not true that every set $\mathcal{M}' \subseteq \mathcal{M}$ of components can be represented as $\mathcal{M}' = \mathcal{M}_c$ for some contract $\mathcal{C}$ and similarly for sets of environments. To highlight this,

$$\text{we assume a family } \mathbf{C} \text{ of expressible contracts.} \qquad (5)$$

Only contracts belonging to this family can be considered. This is made explicit in Table IV.

C. 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 1) implement $\mathcal{C}$ and 2) be able to operate in any environment for $\mathcal{C}$. Hence the following definition for refinement preorder $\preceq$ between contracts: $\mathcal{C}'$ refines $\mathcal{C}$, written $\mathcal{C}' \preceq \mathcal{C}$, if and only if $\mathcal{M}_{\mathcal{C}'} \subseteq \mathcal{M}_{\mathcal{C}}$ and $\mathcal{E}_{\mathcal{C}'} \supseteq \mathcal{E}_{\mathcal{C}}$. As a direct consequence, the following property holds, which justifies the use of the term "refinement" for this relation:

Property 1 (refinement):

1) *Any implementation of $\mathcal{C}'$ is an implementation of $\mathcal{C}$: $M \models^{\mathrm{M}} \mathcal{C}' \Rightarrow M \models^{\mathrm{M}} \mathcal{C}$, and*
2) *Any environment of $\mathcal{C}$ is an environment of $\mathcal{C}'$: $E \models^{\mathrm{E}} \mathcal{C} \Rightarrow E \models^{\mathrm{E}} \mathcal{C}'$.*

The *conjunction* of contracts $\mathcal{C}_1$ and $\mathcal{C}_2$, written $\mathcal{C}_1 \wedge \mathcal{C}_2$, is the greatest lower bound (GLB) of these two contracts with respect to refinement order—we assume such a GLB exists. The intent is to define this conjunction as the intersection of sets of implementations and the union of sets of environments. However, due to (5), not every set of components is the characteristic set of a contract. The best approximation consists in taking the greatest lower bound for the refinement relation. The following immediate properties hold:

Property 2 (shared refinement):

1) *Any contract that refines $\mathcal{C}_1 \wedge \mathcal{C}_2$ also refines $\mathcal{C}_1$ and $\mathcal{C}_2$. Any implementation of $\mathcal{C}_1 \wedge \mathcal{C}_2$ is a shared implementation of $\mathcal{C}_1$ and $\mathcal{C}_2$. Any environment of $\mathcal{C}_1$ or $\mathcal{C}_2$ is an environment of $\mathcal{C}_1 \wedge \mathcal{C}_2$.*
2) *For $\mathbf{C} \subseteq \mathbf{C}$ a subset of contracts, $\bigwedge \mathbf{C}$ is compatible if and only if there exists a compatible $\mathcal{C} \in \mathbf{C}$.*

The conjunction operation formalizes the intuitive notion of a "set of contracts" or a "set of requirements".

D. Contract composition

The objective of contract composition is to ensure that:

1) composing respective implementations of each contract yields an implementation for the composition, and,
2) composing an environment for the resulting composition with an implementation for $\mathcal{C}_2$ yields an environment for $\mathcal{C}_1$ and vice-versa.

These two properties are essential in supporting top-down design by successive decompositions and refinements of contracts. *Contract composition* $\mathcal{C}_1 \otimes \mathcal{C}_2$ is thus defined on top of component composition as indicated in Table IV, with the following assumption in force throughout this meta-theory:

Assumption 1: The following condition holds:

$$\mathcal{C}_1 \otimes \mathcal{C}_2 \quad \in \quad \mathbf{C}_{\mathcal{C}_1 \otimes \mathcal{C}_2}$$

(Note that this assumption is non trivial since it is not true for an arbitrary set $\mathbf{C} \subseteq \mathbf{C}$ that $\bigwedge_{\mathcal{C} \in \mathbf{C}} \mathcal{C} \in \mathbf{C}$.) The condition for $\mathcal{C}_1 \otimes \mathcal{C}_2$ to be defined is that every pair (M_1, M_2) of respective implementations is composable, so that $M_1 \times M_2$ is well defined—recall that component composability is a typing property. Consequently, if E is an environment for $M_1 \times M_2$, E is composable with both M_1 and M_2 and $E \times M_1$ is an environment for M_2 and vice-versa. Referring to the formula defining $\mathcal{C}_1 \otimes \mathcal{C}_2$, denote by $\mathbf{C}_{\mathcal{C}_1 \otimes \mathcal{C}_2} \subseteq \mathbf{C}$ the set of contracts specified in the brackets. The following lemma will be instrumental:

Lemma 1: Let four contracts be such that $\mathcal{C}'_1 \preceq \mathcal{C}_1$, $\mathcal{C}'_2 \preceq \mathcal{C}_2$, and $\mathcal{C}_1 \otimes \mathcal{C}_2$ is well defined. Then, so is $\mathcal{C}'_1 \otimes \mathcal{C}'_2$ and $\mathbf{C}_{\mathcal{C}'_1 \otimes \mathcal{C}'_2} \supseteq \mathbf{C}_{\mathcal{C}_1 \otimes \mathcal{C}_2}$.

Proof: Since $\mathcal{C}_1 \otimes \mathcal{C}_2$ is well defined, it follows that every pair (M_1, M_2) of respective implementations of these contracts is a composable pair of components. Hence, $\mathcal{C}'_1 \otimes \mathcal{C}'_2$ is well defined according to the formula of Table IV. Next, since $\mathcal{C}'_1 \preceq \mathcal{C}_1$ and $\mathcal{C}'_2 \preceq \mathcal{C}_2$ and using Assumption 1, $M_1 \models^{\mathrm{M}} \mathcal{C}'_1$ and $M_2 \models^{\mathrm{M}} \mathcal{C}'_2$ implies $M_1 \models^{\mathrm{M}} \mathcal{C}_1$ and $M_2 \models^{\mathrm{M}} \mathcal{C}_2$; similarly $E \times M_2 \models^{\mathrm{E}} \mathcal{C}_1$ and $E \times M_1 \models^{\mathrm{E}} \mathcal{C}_2$ implies $E \times M_2 \models^{\mathrm{E}} \mathcal{C}'_1$ and $E \times M_1 \models^{\mathrm{E}} \mathcal{C}'_2$. Therefore, replacing, in the big brackets defining the contract composition, $\mathcal{C}_1$ by $\mathcal{C}'_1$ and $\mathcal{C}_2$ by $\mathcal{C}'_2$ can only increase the set $\mathbf{C}_{\mathcal{C}_1 \otimes \mathcal{C}_2}$. □

To conform to the usage, we say that $\mathcal{C}_1$ and $\mathcal{C}_2$ are *compatible* contracts if their composition $\mathcal{C}_1 \otimes \mathcal{C}_2$ is defined and compatible in the sense of Table IV. The following properties are a direct corollary of Lemma 1:

Property 3 (independent implementability): For all contracts $\mathcal{C}_1$, $\mathcal{C}_2$, $\mathcal{C}'_1$ and $\mathcal{C}'_2$, if

1) *$\mathcal{C}_1$ is compatible with $\mathcal{C}_2$,*
2) *$\mathcal{C}'_1 \preceq \mathcal{C}_1$ and $\mathcal{C}'_2 \preceq \mathcal{C}_2$ hold,*

then $\mathcal{C}'_1$ is compatible with $\mathcal{C}'_2$ and $\mathcal{C}'_1 \otimes \mathcal{C}'_2 \preceq \mathcal{C}_1 \otimes \mathcal{C}_2$.

Thus, compatible contracts can be independently refined. This property holds in particular if $\mathcal{C}'_1$ and $\mathcal{C}'_2$ are singletons:

Corollary 1: Compatible contracts can be independently implemented.

Property 3 is fundamental, particularly in top-down design. Top-down incremental design consists in iteratively decomposing a system-level contract $\mathcal{C}$ into sub-system contracts $\mathcal{C}_i, i \in I$ for further independent development. To ensure that independent development will not lead to integration problems, it is enough to verify that $\bigotimes_{i \in I} \mathcal{C}_i \preceq \mathcal{C}$. We insist that, since contracts are purposely abstract and subsystems are not many, the composition of contracts $\mathcal{C}_i$ will not typically result in state explosion.[33]

The following property is essential as it states that contract composition can be performed in any order and changes in architecture (captured by changes in parenthesizing) are allowed:

Property 4 (associativity and commutativity): For all contracts $\mathcal{C}_1$, $\mathcal{C}_2$, $\mathcal{C}_3$ and $\mathcal{C}_4$, if $\mathcal{C}_1$ and $\mathcal{C}_2$ are compatible, $\mathcal{C}_3$ and $\mathcal{C}_4$ are compatible and $\mathcal{C}_1 \otimes \mathcal{C}_2$ is compatible with $\mathcal{C}_3 \otimes \mathcal{C}_4$, then $\mathcal{C}_1$ is compatible with $\mathcal{C}_3$, $\mathcal{C}_2$ is compatible with $\mathcal{C}_4$, $\mathcal{C}_1 \otimes \mathcal{C}_3$ is compatible with $\mathcal{C}_2 \otimes \mathcal{C}_4$, and

$$(\mathcal{C}_1 \otimes \mathcal{C}_2) \otimes (\mathcal{C}_3 \otimes \mathcal{C}_4) \quad = \quad (\mathcal{C}_1 \otimes \mathcal{C}_3) \otimes (\mathcal{C}_2 \otimes \mathcal{C}_4) \qquad (6)$$

[33] This is unlike in *compositional verification*, where $\times_{i \in I} M_i \models^{\mathrm{M}} P$ is to be checked, where M_i are detailed implementations and P is a property. In this case, the composition $\times_{i \in I} M_i$ typically gives raise to state explosion. Techniques have thus been proposed to verify such properties in an incremental way [176], [62], [98], [1], [121].

Proof: To shorten notations, write $\mathcal{C}_{12}$ instead of $\mathcal{C}_1 \otimes \mathcal{C}_2$ and similarly for any subset of $\{1,2,3,4\}$. By Assumption 1 and the associativity and commutativity of component composition, $\mathcal{C}_{1234}$ is characterized by the following two properties, where index i ranges over the set $1 \ldots 4$:

$$\begin{array}{rcl} M_i \models^{\mathrm{M}} \mathcal{C}_i & \Rightarrow & M_1 \times \ldots \times M_4 \models^{\mathrm{M}} \mathcal{C}_{1234} \\ E \models^{\mathrm{E}} \mathcal{C}_{1234} & \Rightarrow & E \times (\times_{j \neq i} M_j) \models^{\mathrm{E}} \mathcal{C}_i \end{array} \tag{7}$$

Observe that (7) is fully symmetric, which proves (6). Next, using the assumptions regarding compatibility, we derive the existence of at least one environment E satisfying the premise of the second implication of (7). Since (7) is fully symmetric, this proves the conclusions of Property 4 regarding compatibility. □

Property 5 (distributivity): If the following contract compositions are all well defined, then the following holds:

$$[(\mathcal{C}_{11} \wedge \mathcal{C}_{21}) \otimes (\mathcal{C}_{12} \wedge \mathcal{C}_{22})] \preceq [(\mathcal{C}_{11} \otimes \mathcal{C}_{12}) \wedge (\mathcal{C}_{21} \otimes \mathcal{C}_{22})] \tag{8}$$

Proof: By Lemma 1, $\mathbf{C}_{(\mathcal{C}_{11} \wedge \mathcal{C}_{21}) \otimes (\mathcal{C}_{12} \wedge \mathcal{C}_{22})} \supseteq \mathbf{C}_{\mathcal{C}_{11} \otimes \mathcal{C}_{12}}$. Taking the GLB of these two sets thus yields $[(\mathcal{C}_{11} \wedge \mathcal{C}_{21}) \otimes (\mathcal{C}_{12} \wedge \mathcal{C}_{22})] \preceq (\mathcal{C}_{11} \otimes \mathcal{C}_{12})$ and similarly for $(\mathcal{C}_{21} \otimes \mathcal{C}_{22})$. Thus, (8) follows. □

The use of distributivity is best illustrated in the following context. Suppose the system under design decomposes into two sub-systems labeled 1 and 2, and each subsystem has two viewpoints associated with it, labeled by another index with values 1 or 2. Contract $(\mathcal{C}_{11} \wedge \mathcal{C}_{21})$ is then the contract associated with sub-system 1 and similarly for sub-system 2. Thus, the left hand side of (8) specifies the set of implementations obtained by, first, implementing each sub-system independently, and then, composing these implementations. Property 5 states that, by doing so, we obtain an implementation of the overall contract obtained by, first, getting the two global viewpoints $(\mathcal{C}_{11} \otimes \mathcal{C}_{12})$ and $(\mathcal{C}_{21} \otimes \mathcal{C}_{22})$, and, then, taking their conjunction. This property supports independent implementation for specifications involving multiple viewpoints. Observe that only refinement, not equality, holds in (8).

E. Quotient

The quotient of two contracts is defined in Table IV. It 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 behaving according to another specification. From its definition in Table IV, we deduce the following property:

Property 6 (quotient): The following holds:

$$\mathcal{C} \preceq \mathcal{C}_1/\mathcal{C}_2 \quad \Leftrightarrow \quad \mathcal{C} \otimes \mathcal{C}_2 \preceq \mathcal{C}_1$$

Proof: Immediate, from the definition. □

F. Discussion

By inspecting Table IV, the different notions can be classified into the following two categories:

- *Primitive notions* that are assumed by the meta-theory. This category comprises: components, component composability and composition.
- *Derived notions* comprise: contract; refinement, conjunction, composition, and quotient; consistency and compatibility for contracts. The derived notions follow from the primitive ones through set theoretic, non effective, definitions.

The meta-theory offers by itself a number of fundamental properties that underpin incremental development and independent implementability. Concrete theories will offer definitions for the primitive notions as well as effective means to implement (or sometimes approximate) the derived notions. *Observers* and then *abstract interpretation* we develop next provide generic approaches to recover effectiveness.

G. Observers

Notion	Observer
$\mathcal{C} = (\mathcal{E}_{\mathcal{C}}, \mathcal{M}_{\mathcal{C}})$	$(b^{\mathsf{E}}_{\mathcal{C}}, b^{\mathsf{M}}_{\mathcal{C}})$
$\mathcal{C} = \mathcal{C}_1 \wedge \mathcal{C}_2$	$b^{\mathsf{E}}_{\mathcal{C}} = b^{\mathsf{E}}_{\mathcal{C}_1} \vee b^{\mathsf{E}}_{\mathcal{C}_2} \ , \ b^{\mathsf{M}}_{\mathcal{C}} = b^{\mathsf{M}}_{\mathcal{C}_1} \wedge b^{\mathsf{M}}_{\mathcal{C}_2}$
$\mathcal{C} = \mathcal{C}_1 \vee \mathcal{C}_2$	$b^{\mathsf{E}}_{\mathcal{C}} = b^{\mathsf{E}}_{\mathcal{C}_1} \wedge b^{\mathsf{E}}_{\mathcal{C}_2} \ , \ b^{\mathsf{M}}_{\mathcal{C}} = b^{\mathsf{M}}_{\mathcal{C}_1} \vee b^{\mathsf{M}}_{\mathcal{C}_2}$
$\mathcal{C} = \mathcal{C}_1 \otimes \mathcal{C}_2$	$b^{\mathsf{E}}_{\mathcal{C}}(E) = \left\{ \begin{array}{c} b^{\mathsf{M}}_{\mathcal{C}_1}(M_1) \wedge b^{\mathsf{M}}_{\mathcal{C}_2}(M_2) \\ \Downarrow \\ b^{\mathsf{E}}_{\mathcal{C}_2}(E \times M_1) \wedge b^{\mathsf{E}}_{\mathcal{C}_1}(E \times M_2) \end{array} \right.$ $b^{\mathsf{M}}_{\mathcal{C}}(M_1 \times M_2) = b^{\mathsf{M}}_{\mathcal{C}_1}(M_1) \wedge b^{\mathsf{M}}_{\mathcal{C}_2}(M_2)$

Table V
Mirroring the algebra of contracts with observers.

A typical obstacle in getting finite (or, more generally, effective) representations of contracts is the occurrence of infinite data types and functions having infinite domains. These can be dealt with by using observers, which originate from the basic notion of test for programs:

Definition 2: Let $\mathcal{C}$ be a contract. An observer *for $\mathcal{C}$ is a pair $(b^{\mathsf{E}}_{\mathcal{C}}, b^{\mathsf{M}}_{\mathcal{C}})$ of non-deterministic boolean functions $\mathcal{M} \mapsto \{\mathrm{F}, \mathrm{T}\}$ called* verdicts*, such that:*

$$\begin{array}{rcl} b^{\mathsf{E}}_{\mathcal{C}}(M) \text{ outputs } \mathrm{F} & \Longrightarrow & M \notin \mathcal{E}_{\mathcal{C}} \\ b^{\mathsf{M}}_{\mathcal{C}}(M) \text{ outputs } \mathrm{F} & \Longrightarrow & M \notin \mathcal{M}_{\mathcal{C}} \end{array} \tag{9}$$

The functions $M \mapsto b^{\mathsf{E}}_{\mathcal{C}}(M)$ and $M \mapsto b^{\mathsf{M}}_{\mathcal{C}}(M)$ being both non-deterministic accounts for the fact that the outcome of a test depends on the stimuli from the environment and possibly results from internal non-determinism of the tested component itself. Note the single-sided implication in (9), which reflects that tests only provide semi-decisions.

Relations and operations of the meta-theory can be mirrored by relations and operations on observers as explained in Table V—we omit the formulas for the quotient as semi-effectiveness of testing makes it useless. Despite GLB and LUB can be mirrored using the formulas of the table, nothing can be said about the relationship of observers for contracts based on the fact that they are in a refinement ordering. Dually nothing can be inferred in terms of their refinement from such a relationship between the observers. Still, the following weaker result holds, which justifies how GLB and LUB are represented using observers in Table V:

Lemma 2: Let $(b^{\mathsf{E}}_{\mathcal{C}}, b^{\mathsf{M}}_{\mathcal{C}})$ be an observer for $\mathcal{C}$ and let $\mathcal{C}' \preceq \mathcal{C}$. Then, any pair $(b^{\mathsf{E}}, b^{\mathsf{M}})$ satisfying $b^{\mathsf{E}} \geq b^{\mathsf{E}}_{\mathcal{C}}$ and $b^{\mathsf{M}} \leq b^{\mathsf{M}}_{\mathcal{C}}$ is an observer for $\mathcal{C}'$.

The following immediate results hold, regarding consistency and compatibility:

Lemma 3:

1) *If $b^{\mathsf{E}}_{\mathcal{C}}(E)$ outputs* F *for all tested environment E, then $\mathcal{C}$ is incompatible;*
2) *If $b^{\mathsf{M}}_{\mathcal{C}}(M)$ outputs* F *for all tested component M, then $\mathcal{C}$ is inconsistent.*

To summarize, Definition 2 provides semi-decision procedures to check whether a component or an environment are valid for a contract. Lemma 3 provides semi-decision procedures for checking consistency and compatibility. Finally, Table V indicates how operations from the contract algebra can be reflected into operations on observers.

Due to the need for exercising all components or environments, using observers for checking consistency or compatibility is still non-effective. For concrete theories exhibiting some notion of "strongest" environment or component for the considered contract, a reinforcement of Lemma 3 will ensure effectiveness.

In Section VI where concrete theories are reviewed, we indicate, for each theory, how observers can be constructed and how Lemma 3 specializes.

H. Bibliographical note

Abstract contract theories and features of our presentation: Our presentation 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. [22] and Chen et al. [59], albeit with deeply different and complementary approaches.

The publication [22] 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 Section VII are pairs of assertions. Sets of legal environments and sets of implementations are associated to contracts. Finally modal contracts are defined by borrowing ideas from modal specifications we discuss in Section VIII. 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 [59] develops the concept of *declarative specification*, which consists of a tuple $\mathcal{P} = (\Sigma^{\rm in}, \Sigma^{\rm out}, T_\Sigma, F_\Sigma)$, where $\Sigma^{\rm in}$ and $\Sigma^{\rm out}$ are input and output alphabets of actions, $\Sigma = \Sigma^{\rm in} \uplus \Sigma^{\rm out}$, 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 [81] and Negulescu [143]. 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 α, even if it will be refused by the component. If α 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 $\mathcal{Q}$ to be used in place of $\mathcal{P}$ it is required that $\mathcal{Q}$ 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.

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.

Synchronous languages [28], [100], [30] are a formalism of choice in dealing with observers. The family of Synchronous Languages comprises mainly the imperative language Esterel [92], [84] and the dataflow languages Lustre [144] and Signal [157]. 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[34]. 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 RT-Builder[35] tool on top of Signal is an example of framework supporting the combination of functional and timing viewpoints while analyzing an application deployed over a virtual architecture (see Section IV-D4). The widely used Simulink/Stateflow[36] tool by The Mathworks offers similar features. One slight drawback is that its mathematical seman-

[34] http://www.esterel-technologies.com/products/scade-suite/

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

[36] http://www.mathworks.com/products/simulink/

tics 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,[37] allows for including physical system models in observers, e.g., as part of the system environment. The same comment holds regarding Modelica.[38] Actually, observers have been proposed and advocated in the context of Lustre and Scade [101], [102], [103], Esterel [45], and Signal [137], [138]. 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 [151], [86], [85] 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:

> *" 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 } |=> (d 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 [2] 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 [18], [17]. 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 [158], [42], that is supported by the tool RATSY [43]. 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 [87]. A survey of using PSL in runtime verification can be found in [85].

Another relevant formalism for building observers consists of the *Live Sequence Charts* (LSC) [64], [107], [105]. LSC are a graphical specification language based on scenarios that is an extension of Message Sequence Charts (MSC)[39]. A typical LSC consists of a prechart and a main chart. The semantics is that whenever the prechart is satisfied in a run of the system, eventually the main chart must also be satisfied. Precharts should not be confused with assumptions, however, they are rather sort of "prerequisites". LSCs are multi-modal in that any construct in the language can be either *cold* or *hot*, with a semantics of "may happen" or "must happen", respectively. If a cold element is violated (say a condition that is not true when reached), this is considered a legal behavior and some appropriate action is taken. Violation of a hot element, however, is considered a violation of the specification and is not allowed to happen in an execution. LSC are executable. The execution of a set of LSCs is called *play out* by the authors [107]. Play out is useful for checking an LSC specification against a property [66]. LSC can be used as (expressive) tests to monitor the execution of a design over non-terminating runs. While doing so, the same rules as above are applied, except that the design controls the emission of events and the LSC specification is used to react in case of violation of the specification. This is manifested by the absence of rules explaining the occurrence of some event at run-time. This procedure is called *play in* by the authors. To conclude, LSCs are well suited to express observers.

VI. Panorama of concrete theories

For the coming series of sections where concrete contract theories are reviewed, the reader is referred to Table IV of Section V, where the meta-theory of contracts is developed. We will illustrate how various concrete contract theories can be developed by specializing the meta-theory.

VII. Panorama: Assume/Guarantee contracts

Our static example of Section IV provided an example of contract specified using Assumptions and Guarantees. 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 specialization and relate them to the existing literature.

[37] http://www.mathworks.com/products/simscape/

[38] https://www.modelica.org/

[39] http://www.sdl-forum.org/MSC/index.htm

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) [117] 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. Then, we develop the other cases.

A. Dataflow A/G contracts

For this simplest variant, all components and contracts involve a same alphabet Σ of variables, possessing identical[40] domain D.

With this simplification, a *component* M identifies with an *assertion*

$$P \quad \subseteq \quad \Sigma \mapsto \; D^* \cup D^\omega \tag{10}$$

i.e., a subset of the set of all finite or infinite behaviors over alphabet Σ; such a behavior associates, to each symbol $x \in \Sigma$, a finite or infinite *flow* of values. The flows are not mutually synchronized and there is no global clock or logical step. We discuss in Section VII-D variants of this framework with more synchronous models of behaviors. For convenience, we feel free to use both set theoretic and boolean notations for the relations and operations on assertions.

Two components are always composable and we define component composition by the intersection of their respective assertions:

$$P_1 \times P_2 \quad = \quad P_1 \wedge P_2 \tag{11}$$

Formulas (10) and (11) define a framework of asynchronous components, with no global clock and no notion of reaction. Instead, a component specifies a relation between the histories of its different flows. When input and output ports are considered as in Section VII-B and components are input/output *functions*, we obtain the model of *Kahn Process Networks* [117] widely used for the mapping of synchronous programs over distributed architectures [159], [160].

Definition 3: A contract *is a pair $\mathcal{C} = (A, G)$ of assertions, called the* assumptions *and the* guarantees. *The set $\mathcal{E}_\mathcal{C}$ of the legal environments for $\mathcal{C}$ collects all components E such that $E \subseteq A$. The set $\mathcal{M}_\mathcal{C}$ of all components implementing $\mathcal{C}$ is defined by $A \times M \subseteq G$.*

Thus, any component M such that $M \leq G \vee \neg A$ is an implementation of $\mathcal{C}$ and $M_\mathcal{C} = G \vee \neg A$ is the maximal implementation. Observe that two contracts $\mathcal{C}$ and $\mathcal{C}'$ with identical input and output alphabets, identical assumptions, and such that $G' \vee \neg A' = G \vee \neg A$, possess identical sets of implementations: $\mathcal{M}_\mathcal{C} = \mathcal{M}_{\mathcal{C}'}$. According to our meta-theory, such two contracts are equivalent. Any contract $\mathcal{C} = (A, G)$ is equivalent to a contract in *saturated form* (A, G') such that $G' \supseteq \neg A$, or, equivalently, $G \vee A = \mathrm{T}$, the true assertion; to exhibit G', just take $G' = G \vee \neg A$. A saturated contract $\mathcal{C} = (A, G)$ is *consistent* if and only if $G \neq \emptyset$ and *compatible* if and only if $A \neq \emptyset$.

Next, for $\mathcal{C}$ and $\mathcal{C}'$ two saturated contracts with identical input and output alphabets,

$$\textit{refinement} \;\; \mathcal{C}' \preceq \mathcal{C} \;\; \text{holds iff} \;\; \left\{ \begin{array}{l} A' \geq A \\ G' \leq G \end{array} \right. \tag{12}$$

Conjunction follows from the refinement relation: for $\mathcal{C}_1$ and $\mathcal{C}_2$ two saturated contracts with identical input and output alphabets: $\mathcal{C}_1 \wedge \mathcal{C}_2 = (A_1 \vee A_2, G_1 \wedge G_2)$.

Focus now on contract *composition* $\mathcal{C} = \mathcal{C}_1 \otimes \mathcal{C}_2$, for two saturated contracts—this is no loss of generality, as we have seen. Contract composition instantiates through the following formulas, cf. (1) in Section IV:

$$\begin{array}{rcl} G & = & G_1 \wedge G_2 \\ A & = & \max \left\{ A \left| \begin{array}{l} A \wedge G_2 \Rightarrow A_1 \\ \text{and} \\ A \wedge G_1 \Rightarrow A_2 \end{array} \right. \right\} \end{array} \tag{13}$$

which satisfies Assumption 1 of the meta-theory. If, furthermore, contracts are saturated, then (13) reformulates as the formulas originally proposed in [29]:

$$\begin{array}{rcl} G & = & G_1 \wedge G_2 \\ A & = & (A_1 \wedge A_2) \vee \neg(G_1 \wedge G_2) \end{array} \tag{14}$$

Observe that the so obtained contract (A, G) is saturated: $G \vee A = (G_1 \wedge G_2) \vee (A_1 \wedge A_2) \vee \neg(G_1 \wedge G_2) = \mathrm{T}$.

No quotient operation is known for Assume/Guarantee contracts.

We finish this section by observing that the two contracts $\mathcal{C}_1$ and $\mathcal{C}'_1$ of Section IV-A1 satisfy $\mathcal{C}'_1 \preceq \mathcal{C}_1$ according to the theory of this section: guarantees are identical but assumptions are relaxed.

B. Capturing exceptions

Referring to the primer of Section IV-A1 and its static system example, dividing by zero may raise an exception. For instance, a careless designer may consider, instead of M_1, a slight variation M'_1 of it where the behaviors are only partially defined:

$$M'_1 \quad : \quad \text{behaviors} \;\; (y \neq 0 \rightarrow z = x/y)$$

leaving what happens if a value 0 is submitted for y unspecified. In practice, $y = 0$ would raise an exception, leading to an unpredictable outcome and possibly a crash unless exception handling is offered as a rescue by the execution platform.

In this section, we show how a mild adjustment of our theory of A/G contracts can capture exceptions and their handling. To simplify, we develop this again for the case of a fixed alphabet Σ. We only present the add-ons with respect to the previous theories, the parts that remain unchanged are not repeated.

Since exceptions are undesirable events caused by the component itself and not by its inputs—for our simple example, the

[40] This is only an assumption intended to simplify the notations. It is by no means essential.

exception is the improper handling of the division by zero—we need to include inputs and outputs as part of our framework for components.

A *component* is thus a tuple $M = (\Sigma^{\rm in}, \Sigma^{\rm out}, P)$, where $\Sigma = \Sigma^{\rm in} \cup \Sigma^{\rm out}$ is the decomposition of alphabet Σ into its *inputs* and *outputs*, and $P \subseteq (\Sigma \mapsto (D^* \cup D^\omega))$ is an assertion. Whenever convenient, we shall denote by $\Sigma^{\rm in}_M$, Q_M, P_M, etc., the items defining component M. Components M_1 and M_2 are *composable* if $\Sigma^{\rm out}_1 \cap \Sigma^{\rm out}_2 = \emptyset$. If so, then $M_1 \times M_2 = (\Sigma^{\rm in}, \Sigma^{\rm out}, P)$ is well defined, with $\Sigma^{\rm out} = \Sigma^{\rm out}_1 \cup \Sigma^{\rm out}_2$, $\Sigma^{\rm in} = \Sigma - \Sigma^{\rm out}$, and $P = P_1 \cap P_2$.

Let us now focus on exceptions. To capture improper response (leading to a "crash"), we distinguish a special element $fail \in D$. We assume that crash is not revertible, i.e., in any behavior of the component, any variable remains at $fail$ when it reaches that value. Referring to the static example of Section IV, we would then set $x/0 := fail$ for any x. We are now ready to formalize what it means, for a component, to be free of exception:

Definition 4: A component M is free of exception *if:*

1) *It accepts all inputs:*

$$\mathbf{pr}_{\Sigma^{\rm in}_M}(P_M) = \Sigma^{\rm in}_M \mapsto (D^* \cup D^\omega)$$

2) *It does not cause by itself the occurrence of fail in its behaviors: for any behavior $\sigma \in P_M$, if the projection $\mathbf{pr}_{\Sigma^{\rm in}_M}(\sigma)$ of σ to the input alphabet $\Sigma^{\rm in}_M$ does not involve fail, then neither does σ.*

Exception freeness defined in this way is such that, if M_1 and M_2 are both composable and free of exception, then $M_1 \times M_2$ is also free of exception. Hence, we are free to restrict our universe of components to *exception free* components—thus, Definition 4 defines our family of components. A/G contracts are re-defined accordingly:

Definition 5: A contract *is a tuple $\mathcal{C} = (\Sigma^{\rm in}, \Sigma^{\rm out}, A, G)$, where $\Sigma^{\rm in}$ and $\Sigma^{\rm out}$ are the input and output alphabets and A and G are assertions over Σ, called the* assumptions *and the* guarantees*. The set $\mathcal{E}_c$ of the legal environments for $\mathcal{C}$ are all free of exception components E such that $\Sigma^{\rm in}_E = \Sigma^{\rm out}_{\mathcal{C}}$, $\Sigma^{\rm out}_E = \Sigma^{\rm in}_{\mathcal{C}}$, and $P_E \subseteq A$. The set $\mathcal{M}_c$ of all components implementing $\mathcal{C}$ is defined by: M is free of exception, $\Sigma^{\rm in}_M = \Sigma^{\rm in}_{\mathcal{C}}$ and $\Sigma^{\rm out}_M = \Sigma^{\rm out}_{\mathcal{C}}$ and $P_{E \times M} \subseteq G$ for every environment E of $\mathcal{C}$.*

Focus now on the issue of consistency and compatibility, for contracts. The following holds:

Property 7: Let $\mathcal{C} = (\Sigma^{\rm in}, \Sigma^{\rm out}, A, G)$ be a contract satisfying the following conditions:

$$\mathbf{pr}_{\Sigma^{\rm in}_{\mathcal{C}}}(G) = \Sigma^{\rm in}_M \mapsto (D^* \cup D^\omega) \tag{15}$$

$$\mathbf{pr}_{\Sigma^{\rm out}_{\mathcal{C}}}(A) = \Sigma^{\rm out}_M \mapsto (D^* \cup D^\omega) \tag{16}$$

$$fail \quad \textit{does not occur in } G \wedge A \tag{17}$$

Then, $\mathcal{C}$ is consistent and compatible.

Proof: By condition (15), the component $M = (\Sigma^{\rm in}, \Sigma^{\rm out}, G)$ satisfies condition 4 of Definition 4. It may not satisfy condition 2, however. To achieve this we must modify, in M, the behaviors not belonging to A to avoid $fail$ to occur. Preserving M on A will ensure that implementation of $\mathcal{C}$ is preserved. To get M', replace any behavior $\sigma \in G \wedge \neg A$ by a behavior σ' such that $\mathbf{pr}_{\Sigma^{\rm in}_{\mathcal{C}}}(\sigma') = \mathbf{pr}_{\Sigma^{\rm in}_{\mathcal{C}}}(\sigma)$ and $\mathbf{pr}_{\Sigma^{\rm out}_{\mathcal{C}}}(\sigma') \not\ni fail$. Component M' obtained in this way is free of exception and implements $\mathcal{C}$, showing that $\mathcal{C}$ is consistent. Consider next the component $E = (\Sigma^{\rm out}, \Sigma^{\rm in}, A)$. If E is free of exception, then it is an environment for $\mathcal{C}$. If this is not the case, then we can modify E on $A \wedge \neg G$ as we did for obtaining M'. This yields an environment E' for $\mathcal{C}$, showing that $\mathcal{C}$ is compatible. □

Conditions (15) and (16) express that assumptions and guarantees are both input enabled. Condition (17) is the key one. Observe that, now, contract $\mathcal{C}'_1$ of Section IV-A1 is inconsistent since it has no implementation—implementations must be free of exception. In turn, contract $\mathcal{C}_1$ is consistent. This is in contrast to the theory of Section VII-A, where both contracts were considered consistent (crashes were not ruled out). Indeed, contract $\mathcal{C}_1$ of Section IV-A1 satisfies the conditions of Property 7, whereas $\mathcal{C}'_1$ does not. Addressing exceptions matters.

C. Dealing with variable alphabets

Since contracts aim at capturing incomplete designs, we cannot restrict ourselves to a fixed alphabet—it is not known in advance what the actual alphabet of the complete design will be. Thus the simple variants of Sections VII-A and VII-B have no practical relevance and we must extend them to dealing with variable alphabets. In particular, components will now be pairs $M = (\Sigma_M, P_M)$, where Σ_M is the alphabet of M and P_M is an assertion over Σ_M. Similarly, contracts are triples $\mathcal{C} = (\Sigma_{\mathcal{C}}, A_{\mathcal{C}}, G_{\mathcal{C}})$, where assumptions $A_{\mathcal{C}}$ and guarantees $G_{\mathcal{C}}$ are assertions over alphabet $\Sigma_{\mathcal{C}}$.

Key to dealing with variable alphabet this is the operation of *alphabet equalization* that we introduce next. For P an assertion over alphabet Σ and $\Sigma' \subseteq \Sigma$, we consider its *projection* $\mathbf{pr}_{\Sigma'}(P)$ over Σ', which is simply the set of all restrictions, to Σ', of all behaviors belonging to P. We will also need the *inverse projection* $\mathbf{pr}^{-1}_{\Sigma''}(P)$, for $\Sigma'' \supseteq \Sigma$, which is the set of all behaviors over Σ'' projecting, to Σ, as behaviors of P. For $(\Sigma_i, P_i), i = 1, 2$, we call *alphabet equalization* of (Σ_1, P_1) and (Σ_2, P_2) the two assertions $(\Sigma, \mathbf{pr}^{-1}_{\Sigma}(P_i)), i = 1, 2$ where $\Sigma = \Sigma_1 \cup \Sigma_2$.

We also need to define alphabet equalization when the alphabet Σ decomposes as $\Sigma = \Sigma^{\rm in} \uplus \Sigma^{\rm out}$. Equalizing the above decomposition to a larger alphabet $\Sigma'' \supseteq \Sigma$ yields $\Sigma''^{\rm out} = \Sigma^{\rm out}$, whence $\Sigma''^{\rm in} = \Sigma'' \setminus \Sigma''^{\rm out}$ follows.

Alphabet equalization serves as a preparation step to reuse the framework of Section VII-A when alphabets are variable.

This being defined, all operations and relations introduced in Section VII-A are extended to the case of variable alphabets by 1) applying alphabet equalization to the involved assertions, and, 2) reusing the operation or relation as introduced in Section VII-A.

As pointed out in [29], this generalization yields a contract theory that is a valid instantiation of the meta-theory (up to the missing quotient). It is not satisfactory from the practical standpoint, however. The reason is that the conjunction of two contracts with disjoint alphabets yields a trivial assumption T, which is very demanding—any environment must be accommodated—and does not reflect the intuition (more on this will be discussed in Section VIII-D.

To summarize, Sections VII-A, VII-B, and VII-C, together define a framework for *asynchronous Assume/Guarantee contracts* where components are of Kahn Process Network type.

D. Synchronous A/G contracts

We obtain variants of this framework of Assume/Guarantee contracts by changing the concrete definition of what an assertion is, and possibly revisiting what the component composition is. We can redefine assertions as

$$P \quad \subseteq \quad (\Sigma \mapsto D)^* \cup (\Sigma \mapsto D)^\omega . \tag{18}$$

Compare (18) with (10). In both cases, assertions are sets of behaviors. With reference to (10), behaviors were tuples of finite or infinite flows, one for each symbol of the alphabet. In contrast, in (18), behaviors are finite or infinite sequences of *reactions*, which are the assignment of a value to each symbol of the alphabet. By having a distinguished symbol $\perp \in D$ to model the *absence* of an actual data, we get the multiple-clocked synchronous model used by synchronous languages [30]. Definition (18) for assertions correspond to the synchronous model of computation, whereas (10) corresponds to the Kahn Process Network type of model [117], [142]. The material of Sections VII-A, VII-B, and VII-C, can be adapted to this new model of component composition, thus yielding a framework of *synchronous Assume/Guarantee contracts.*

E. Observers

The construction of observers for this case is immediate. We develop it for the most interesting case in which exceptions are handled, see Sections VII-A and VII-B. Let P be an assertion according to (10). P defines a verdict b_P by setting

$$b_P(\sigma) = \mathrm{T} \ \text{ if and only if } \sigma \in P \tag{19}$$

Observers must return their verdict in some finite amount of time. Hence, on-line interpretation based on Definition 3 is appropriate. With this interpretation, for $\mathcal{C} = (\Sigma^{\text{in}}, \Sigma^{\text{out}}, A, G)$ a contract, its associated observer is obtained as follows, with reference to Definition 2:

- $b^{\mathsf{E}}_{\mathcal{C}}(E)$ is performed by drawing non-deterministically a behavior σ of E and then evaluating $b_A(\sigma)$.
- $b^{\mathsf{M}}_{\mathcal{C}}(M)$ is performed by drawing non-deterministically a behavior σ of M and then evaluating $b_A(\sigma) \Rightarrow b_M(\sigma)$.

Lemma 3 for generic observers specializes to the following, effective, semi-decision procedure:

Lemma 4:

1) *If b_A outputs* F*, then $\mathcal{C}$ is incompatible;*
2) *If $b_A \Rightarrow b_G$ outputs* F*, then $\mathcal{C}$ is inconsistent.*

F. Discussion

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

Regarding exceptions, the special value *fail* that we introduced to capture exceptions is not something that is given in practice. Value *fail* may subsume various run time errors. Alternatively, for the synchronous Assume/Guarantee contracts, *fail* can capture the failure of a component to be input enabled (able, in each reaction, to accept any tuple of inputs).

In its present form, the framework of Assume/Guarantee contracts (synchronous or asynchronous), suffers from the need to manipulate negations of assertions, an operation that is generally not effective, except if the framework is restricted to finite state automata and finite alphabets of actions. For general frameworks, using observers or abstract interpretation can mitigate this in part.

G. 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 [122], [58], [136], [14], [61]. A/G-contracts were advocated by the SPEEDS project [29]. They were further experimented in the framework of the CESAR project [63], with the additional consideration of *weak* and *strong* assumptions. The theory developed in [29] turns out to be closest to this presentation; still, exceptions were not handled in [29]. The presentation developed in this paper clarifies the design choices in A/G-contract theories.

Inspired by [126], another form for A/G-contract was proposed by [97], [99] when designs are expressed using the BIP programming language [41], [174]. 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.

Regarding extensions, a notion of *contract for real-time interfaces* is proposed in [40]. Sets of tasks are associated to components which are individually schedulable on a processor. An interface for a component is an ω-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 [145], 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/G-contracts have been extended to a stochastic setting by Delahaye et al. [75], [76], [77]. 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 α_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 $\alpha_1 + \alpha_2 - 1$.

Features of our presentation: This presentation of A/G-contracts is new in may 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 [29]. Also, this allowed us to handle exceptions properly.

VIII. PANORAMA: INTERFACE THEORIES

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*. Interface theories generally use (a mild variation of) Lynch Input/Output Automata [135], [134] as their framework for components and environments. As a prerequisite, we thus recall the background on Input/Output Automata, i/o-automata for short.

A. Components as i/o-automata

An *i/o-automaton* is a tuple $M = (\Sigma^{\rm in}, \Sigma^{\rm out}, Q, q_0, \rightarrow)$, where

- $\Sigma^{\rm in}$ and $\Sigma^{\rm out}$ are disjoint finite input and output alphabets; set $\Sigma = \Sigma^{\rm in} \cup \Sigma^{\rm out}$;
- 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*; 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 $q \xrightarrow{\alpha} q'$.

By concatenation, transition relation $\rightarrow$ extends to a relation $\rightarrow^*$ on $Q \times \Sigma^* \times Q$, where Σ^* is the set of all finite words over Σ. Say that a state q' is *reachable* from q if there exists some word m such that $q \xrightarrow{m}{}^* q'$. To considerably simplify the development of the theory, we restrict ourselves to

$$\begin{array}{l}\textit{deterministic} \text{ i/o-automata, i.e., such that:} \\ q \xrightarrow{\alpha} q_1 \text{ and } q \xrightarrow{\alpha} q_2 \text{ implies } q_2 = q_1.\end{array} \tag{20}$$

Two i/o-automata M_1 and M_2 having identical alphabet Σ are composable if the usual input/output matching condition holds: $\Sigma_1^{\rm out} \cap \Sigma_2^{\rm out} = \emptyset$ and their composition $M = M_1 \times M_2$ is given by

$$\begin{array}{rcl}\Sigma^{\rm in} & = & \Sigma_1^{\rm in} \cap \Sigma_2^{\rm in} \\ \Sigma^{\rm out} & = & \Sigma_1^{\rm out} \cup \Sigma_2^{\rm out} \\ Q & = & Q_1 \times Q_2 \text{ and } q_0 = (q_{1,0}, q_{2,0}),\end{array}$$

and the transition relation $\rightarrow$ is given by

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

An i/o-automaton is said *closed* whenever $\Sigma^{\rm in} = \emptyset$ and *open* otherwise. 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

$$\forall \alpha, q_2' \text{ such that } q_2 \xrightarrow{\alpha}_2 q_2' \;\; \Rightarrow \left\{ \begin{array}{l} q_1 \xrightarrow{\alpha}_1 q_1' \\ \text{and } q_2' \leq q_1' \end{array} \right. \tag{21}$$

Say that

$$M_1 \textit{ simulates } M_2, \text{ written } M_2 \leq M_1, \text{ if } q_{2,0} \leq q_{1,0}. \tag{22}$$

Observe that simulation relation (21,22) does not distinguish inputs from outputs neither it distinguishes the component from its environment. It is the classical simulation relation meant for closed systems. Due to condition (20) of determinism, simulation coincides with language inclusion.

Variable alphabet is again dealt with using the mechanism of alphabet equalization. For $M = (\Sigma^{\rm in}, \Sigma^{\rm out}, Q, q_0, \rightarrow)$ an i/o-automaton and $\Sigma' \supset \Sigma$, we define

$$M^{\uparrow \Sigma'} \;\; = \;\; (\Sigma^{\rm in} \cup (\Sigma' \setminus \Sigma), \Sigma^{\rm out}, Q, q_0, \rightarrow')$$

where $\rightarrow'$ is obtained by adding, to $\rightarrow$, for each state and each added action, a self-loop at this state labeled with this action.

Components—and consequently environments—for use in interface theories will be *receptive i/o-automata* (also termed *input enabled*), i.e., they should react by proper response to any input stimulus in any state:[41]

$$M \text{ is } \textit{receptive} \text{ iff } \forall q \in Q, \forall \alpha \in \Sigma^{\text{in}} : \; q \xrightarrow{\alpha} \tag{23}$$

Receptiveness is stable under parallel composition.

B. 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 Σ. We begin with the definition of Interface Automata which are possibly non-receptive i/o-automata. Moreover we give their interpretation as contracts according to the meta-theory.

Definition 6: An Interface Automaton *is a tuple* $\mathcal{C} = (\Sigma^{\text{in}}, \Sigma^{\text{out}}, Q, q_0, \rightarrow)$ *where* $\Sigma^{\text{in}}, \Sigma^{\text{out}}, Q, \rightarrow$ *are as in i/o-automata.*

The initial state q_0*, however, may not satisfy* $q_0 {\in} Q$*. If* $q_0 {\in} Q$ *holds, Interface Automaton* $\mathcal{C}$ *defines a contract by fixing a pair* $(\mathcal{E}_{\mathcal{C}}, \mathcal{M}_{\mathcal{C}})$ *as follows:*

- *The set* $\mathcal{E}_{\mathcal{C}}$ *of the* legal environments *for* $\mathcal{C}$ *collects all components* E *such that:*
  1) $\Sigma_E^{\text{in}} = \Sigma^{\text{out}}$ *and* $\Sigma_E^{\text{out}} = \Sigma^{\text{in}}$*. Thus,* E *and* $\mathcal{C}$*, seen as i/o-automata, are composable and* $E \times \mathcal{C}$ *is closed;*
  2) *For any output action* $\alpha \in \Sigma^{\text{in}}$ *of environment* E *such that* $q_E \xrightarrow{\alpha}_E$ *and any reachable state* (q_E, q) *of* $E \times \mathcal{C}$*, then* $(q_E, q) \xrightarrow{\alpha}_{E \times \mathcal{C}}$ *holds.*

  There exists a unique maximal *environment* $E_{\mathcal{C}} \in \mathcal{E}_{\mathcal{C}}$*, in that* $E_{\mathcal{C}} \times \mathcal{C}$ *simulates* $E \times \mathcal{C}$ *in the sense of (22) for any* $E \in \mathcal{E}_{\mathcal{C}}$*;* $E_{\mathcal{C}} = (\Sigma^{\text{out}}, \Sigma^{\text{in}}, Q \cup \{\top\}, q_0, \rightarrow_{E_{\mathcal{C}}})$*, where:*
  (a) *the restriction, to* $Q \times \Sigma \times Q$*, of transition relation* $\rightarrow_{E_{\mathcal{C}}}$*, coincides with* $\rightarrow$*;*
  (b) $\forall q \in Q$*:* $q \xrightarrow{\alpha}_{E_{\mathcal{C}}} \top$ *holds if and only if* $\neg[q \xrightarrow{\alpha}\,]$ *and* $\alpha \in \Sigma^{\text{out}}$*;*
  (c) $\top \xrightarrow{\alpha}_{E_{\mathcal{C}}} \top$ *holds for any* $\alpha \in \Sigma$*.*
- *The set* $\mathcal{M}_{\mathcal{C}}$ *of the* implementations *of* $\mathcal{C}$ *collects all components* M *such that i/o-automaton* $\mathcal{C}$ *simulates* $E_{\mathcal{C}} {\times} M$ *in the sense of (22).*

Condition 2) means that environment E is only willing to emit an output if it is accepted as an input by $\mathcal{C}$ in the composition $E \times \mathcal{C}$. Regarding the existence and uniqueness of $E_{\mathcal{C}}$, observe first that conditions (a) and (b) together imply that $E_{\mathcal{C}}$ is receptive. Also, by (a) and (b), $E_{\mathcal{C}} \times \mathcal{C}$ is strictly identical with $(\emptyset, \Sigma, Q, q_0, \rightarrow)$, i.e., it is obtained from $\mathcal{C}$, seen as an i/o-automaton, by simply turning inputs to outputs. Consequently, Condition 2) holds and, thus, $E_{\mathcal{C}} \in \mathcal{E}_{\mathcal{C}}$. By (b) and (c), $E_{\mathcal{C}}$ is maximal.

The above definition of Interface Automata is heterodox, compare with the original references [72], [5]. Definition 6 introduces the two sets $\mathcal{E}_{\mathcal{C}}$ and $\mathcal{M}_{\mathcal{C}}$, 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 6 allows to cast Interface Automata in the framework of the meta-theory of Table IV.

Corresponding relations and operations must be instantiated and we do this next. As a first, immediate, result:

Lemma 5: $q_0 \in Q$ *is the necessary and sufficient condition for* $\mathcal{C}$ *to be both consistent and compatible in the sense of the meta-theory, i.e.,* $\mathcal{E}_{\mathcal{C}} \neq \emptyset$ *and* $\mathcal{M}_{\mathcal{C}} \neq \emptyset$*.*

Refinement and conjunction: Contract *refinement* as defined in Table IV is equivalent to *alternate simulation* [10], defined as follows: for $\mathcal{C}_i, i = 1, 2$ two contracts, say that two respective states $q_i, i = 1, 2$ are in alternate simulation, written $q_2 {\preceq} q_1$, if

$$\begin{aligned} \forall \alpha \in \Sigma_2^{\text{out}}, q_2' \text{ s.t. } q_2 \xrightarrow{\alpha}_2 q_2' \quad &\Rightarrow \quad \begin{cases} \alpha \in \Sigma_1^{\text{out}} \\ q_1 \xrightarrow{\alpha}_1 q_1' \\ q_2' {\preceq} q_1' \end{cases} \\ \forall \alpha \in \Sigma_1^{\text{in}}, q_1' \text{ s.t. } q_1 \xrightarrow{\alpha}_1 q_1' \quad &\Rightarrow \quad \begin{cases} \alpha \in \Sigma_2^{\text{in}} \\ q_2 \xrightarrow{\alpha}_2 q_2' \\ q_2' {\preceq} q_1' \end{cases} \end{aligned} \tag{24}$$

Say that $\mathcal{C}_2$ *refines* $\mathcal{C}_1$, written $\mathcal{C}_2 {\preceq} \mathcal{C}_1$, if $q_{2,0} {\preceq} q_{1,0}$. The first condition of (24) reflects the inclusion $\mathcal{M}_{\mathcal{C}_2} \subseteq \mathcal{M}_{\mathcal{C}_1}$, whereas the second condition of (24) reflects the opposite inclusion $\mathcal{E}_{\mathcal{C}_2} \supseteq \mathcal{E}_{\mathcal{C}_1}$. Alternate simulation can be effectively checked, see [70] for issues of computational complexity. Unfortunately, no simple formula for the conjunction of contracts is known. See [83] for the best results in this direction.

Parallel composition and quotient: Contract *composition* $\mathcal{C}_2 {\otimes} \mathcal{C}_1$, as defined in the meta-theory, is effectively computed as follows, for $\mathcal{C}_i$ two Interface Automata satisfying the conditions of Lemma 5. Consider, as a first candidate for contract composition, the composition $\mathcal{C}_2 {\times} \mathcal{C}_1$ where $\mathcal{C}_i, i = 1, 2$ are seen as i/o-automata. This first guess does not work because of the condition involving environments in the contract composition of the meta-theory. More precisely, by the meta-theory we should have

$$E \models^{\text{E}} \mathcal{C} \Rightarrow [E {\times} M_2 \models^{\text{E}} \mathcal{C}_1 \text{ and } E {\times} M_1 \models^{\text{E}} \mathcal{C}_2]$$

which requires $\forall \alpha \in \Sigma_i^{\text{out}}$: $q_i \xrightarrow{\alpha}_i \;\Rightarrow\; (q_1, q_2) \xrightarrow{\alpha}_{\mathcal{C}_2 \times \mathcal{C}_1}$. Pairs (q_1, q_2) not satisfying this are called *illegal*. In words, a pair of states (q_1, q_2) is illegal when one Interface Automaton wants to submit an output whereas the other one refuses to accept it—referring to Assume/Guarantee contracts, one could interpret this as a mismatch of assumptions and guarantees in this pair of interfaces. Illegal pairs must then be pruned away. Pruning away illegal pairs from $\mathcal{C}_2 {\times} \mathcal{C}_1$ together with corresponding incoming transitions may cause other illegal pairs to occur. The latter must also be pruned away, until fixpoint occurs. The result is the contract composition $\mathcal{C}_2 {\otimes} \mathcal{C}_1$.

[41] In fact, receptiveness is assumed in the original notion of i/o-automaton by Nancy Lynch [135], [134]. We use here a relaxed version of i/o-automaton for reasons that will become clear later.

As a result of this pruning, it may be the case that the resulting contract has empty set of states, which, by Lemma 5, expresses that the resulting composition of the two interfaces is inconsistent and incompatible—in the original literature on Interface Automata [72], [5] it is said that the two interfaces $\mathcal{C}_1$ and $\mathcal{C}_2$ are incompatible.

In [39], incremental design of deterministic Interface Automata is studied. Let $\mathcal{C}^{\downarrow}$ be the interface $\mathcal{C}$ with input and output actions interchanged. Given two Interface Automata $\mathcal{C}_1$ and $\mathcal{C}_2$, the greatest interface compatible with $\mathcal{C}_2$ such that their composition refines $\mathcal{C}_1$ is given by $(\mathcal{C}_2 \parallel {\mathcal{C}_1}^{\downarrow})^{\downarrow}$. Hence, the part regarding quotient in the meta-theory is correctly addressed for *deterministic* Interface Automata.

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.[42] 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.

C. 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 VIII-G. As Interface Automata, Modal Interfaces use receptive i/o-automata as their components and environments. The presentation of Modal Interfaces we develop here is aligned with our meta-theory and, thus, differs from classical presentations. Again, we begin with the case of a fixed alphabet Σ.

Definition 7: We call Modal Interface *a tuple of the form* $\mathcal{C} = (\Sigma^{\mathrm{in}}, \Sigma^{\mathrm{out}}, Q, q_0, \rightarrow, \dashrightarrow)$*, where* $\Sigma^{\mathrm{in}}, \Sigma^{\mathrm{out}}, Q, q_0$ *are as in Interface Automata and* $\rightarrow, \dashrightarrow \subseteq Q \times \Sigma \times Q$ *are two transition relations, called* must *and* may*, respectively.*

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

$$\begin{array}{rcl} \mathcal{C}^{must} & = & (\Sigma^{\mathrm{in}}, \Sigma^{\mathrm{out}}, Q, q_0, \rightarrow) \\ \text{and } \mathcal{C}^{may} & = & (\Sigma^{\mathrm{in}}, \Sigma^{\mathrm{out}}, Q, q_0, \dashrightarrow). \end{array}$$

A Modal Interface is deterministic *if $\mathcal{C}^{may}$ is deterministic in the sense of 20. $\mathcal{C}$ defines a contract by fixing a pair $(\mathcal{E}_{\mathcal{C}}, \mathcal{M}_{\mathcal{C}})$ as follows:*

The set $\mathcal{E}_{\mathcal{C}}$ of the legal environments *for $\mathcal{C}$ collects all components E such that:*

1) *$\Sigma_E^{\mathrm{in}} = \Sigma^{\mathrm{out}}$ and $\Sigma_E^{\mathrm{out}} = \Sigma^{\mathrm{in}}$; consequently, E and $\mathcal{C}^{must}$, seen as i/o-automata, are composable and $E \times \mathcal{C}^{must}$ is closed; the same holds with $\mathcal{C}^{may}$ in lieu of $\mathcal{C}^{must}$;*
2) *For any $\alpha \in \Sigma^{\mathrm{in}}$ such that $q_E \xrightarrow{\alpha}_E$ and any reachable state (q_E, q) of $E \times \mathcal{C}^{may}$: $(q_E, q) \xrightarrow{\alpha}_{E \times \mathcal{C}^{must}}$.*

There exists a unique maximal environment $E_{\mathcal{C}} \in \mathcal{E}_{\mathcal{C}}$, in that $E_{\mathcal{C}} \times \mathcal{C}^{may}$ simulates $E \times \mathcal{C}^{may}$ in the sense of (22) for any $E \in \mathcal{E}_{\mathcal{C}}$; $E_{\mathcal{C}} = (\Sigma^{\mathrm{out}}, \Sigma^{\mathrm{in}}, Q \cup \{\top\}, q_0, \rightarrow_{E_{\mathcal{C}}})$, where:

(a) *the restriction, to $Q \times \Sigma^{\mathrm{in}} \times Q$, of transition relation $\rightarrow_{E_{\mathcal{C}}}$, coincides with $\rightarrow$; the restriction, to $Q \times \Sigma^{\mathrm{out}} \times Q$, of transition relation $\rightarrow_{E_{\mathcal{C}}}$, coincides with $\dashrightarrow$;*
(b) *for any $q \in Q$, $q \xrightarrow{\alpha}_{E_{\mathcal{C}}} \top$ holds if and only if $\neg[q \overset{\alpha}{\dashrightarrow}]$ and $\alpha \in \Sigma^{\mathrm{out}}$;*
(c) *$\top \xrightarrow{\alpha}_{E_{\mathcal{C}}} \top$ holds for any $\alpha \in \Sigma$.*

The set $\mathcal{M}_{\mathcal{C}}$ of the implementations *of $\mathcal{C}$ collects all components M such that:*

3) *$E_{\mathcal{C}} \times \mathcal{C}^{may}$ simulates $E_{\mathcal{C}} \times M$ in the sense of (22);*
[*only* may *transitions are allowed for $E_{\mathcal{C}} \times M$;*]
4) *for any reachable state (q_E, q_M) of $E_{\mathcal{C}} \times M$ and (q_E, q) of $E_{\mathcal{C}} \times \mathcal{C}^{may}$ such that $(q_E, q_M) \leq (q_E, q)$, and any action $\alpha \in \Sigma_M^{\mathrm{out}}$ such that $q \overset{\alpha}{\rightarrow}$, then $(q_E, q_M) \xrightarrow{\alpha}_{E_{\mathcal{C}} \times M}$*
[must *transitions are mandatory in $E_{\mathcal{C}} \times M$.*]

The first paragraph is a verbatim of the original definition of Modal Interfaces [166]. Definition 7 introduces the two sets $\mathcal{E}_{\mathcal{C}}$ and $\mathcal{M}_{\mathcal{C}}$, whereas the classical theory of Modal Interfaces considers and develops a different notion of *model* (often also called “implementation”, which is unfortunate). As for Interface Automata in Section VIII-B, the handling of initial state q_0 is heterodox. Our Definition 7 allows to cast Modal Interfaces in the framework of the meta-theory.

To justify the existence and uniqueness of $E_{\mathcal{C}}$, observe first that conditions (a) and (b) together imply that $E_{\mathcal{C}}$ is receptive. Condition (2) follows from (a). Maximality in the sense of (22) follows from (b) and (c).

Consistency and Compatibility: We begin with consistency. Say that state $q \in Q$ of $\mathcal{C}$ is *consistent* if $q \overset{\alpha}{\rightarrow}$ implies $q \overset{\alpha}{\dashrightarrow}$, otherwise we say that it is *inconsistent*. Assume that $\mathcal{C}$ has some inconsistent state $q \in Q$, meaning that, for some action α, $q \overset{\alpha}{\rightarrow}$ holds but $q \overset{\alpha}{\dashrightarrow}$ does not hold. By conditions 3) and 4) of Definition 7, for any environment E and any implementation M of $\mathcal{C}$, no state (q_E, q_M) of $E \times M$ satisfies $(q_E, q_M) \leq q$. Hence, all may transitions leading to q can be deleted from $\mathcal{C}$ without changing $\mathcal{M}_{\mathcal{C}}$. Performing this makes state q unreachable in $\mathcal{C}^{may}$, thus Condition 2 of Definition 7 is relaxed and the set of environments is possibly augmented. Since we have removed may transitions, some more states have possibly become inconsistent. So, we must repeat the same procedure. Since the number of states is finite, this procedure eventually reaches a fixpoint. At fixpoint, the set Q of states partition as $Q = Q_{\mathrm{con}} \uplus Q_{\mathrm{incon}}$, where Q_{incon} collects all states that were or became inconsistent as a result of this procedure, and Q_{con} only collects consistent states. In addition, since the fixpoint has been reached, Q_{incon} is unreachable from Q_{con}. As a final step, we delete Q_{incon} and call $[\mathcal{C}]$ the so obtained contract. The following result holds:

Lemma 6:

[42] The inverse projection of an i/o-automaton is simply achieved by adding, in each state, a self-loop for each missing symbol.

1) *Modal Interface* $[\mathcal{C}]$ *satisfies*

$$\mathcal{M}_{[\mathcal{C}]} = \mathcal{M}_{\mathcal{C}} \quad \textit{and} \quad \mathcal{E}_{[\mathcal{C}]} \supseteq \mathcal{E}_{\mathcal{C}} \tag{25}$$

where the inclusion is strict unless $\mathcal{C}$ *possesses no inconsistent state.*

2) $[\mathcal{C}]$ *offers the smallest set of environments among the Modal Interfaces satisfying (25).*

3) $\mathcal{C}$ *is consistent and compatible if and only if* $Q_{\rm con} \ni q_0$.

In the sequel, unless otherwise specified, we will only consider Modal Interfaces that are consistent and compatible.

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.

For $\mathcal{C}$ a Modal Interface and $q{\in}Q$ a state of it, introduce the following *may* and *must* sets:

$$\begin{array}{rcl} may(q) & = & \{\alpha \in \Sigma \mid q \overset{\alpha}{\dashrightarrow} \} \\ must(q) & = & \{\alpha \in \Sigma \mid q \overset{\alpha}{\to} \} \end{array}$$

Definition 8 (modal refinement): Let $\mathcal{C}_i, i = 1,2$ *be two Modal Interfaces and let* q_i *be a state of* $\mathcal{C}_i$, *for* $i = 1,2$. *Say that* q_2 refines q_1, *written* $q_2 \preceq q_1$, *if:*

$$\left\{ \begin{array}{rcl} may_2(q_2) & \subseteq & may_1(q_1) \\ must_2(q_2) & \supseteq & must_1(q_1) \end{array} \right.$$

$$\textit{and } \forall \alpha \in \Sigma : \left\{ \begin{array}{l} q_1 \overset{\alpha}{\dashrightarrow}_1 q_1' \\ q_2 \overset{\alpha}{\dashrightarrow}_2 q_2' \end{array} \right. \Longrightarrow q_2' \preceq q_1'$$

Say that $\mathcal{C}_2 \preceq \mathcal{C}_1$ *if* $q_{2,0} \preceq q_{1,0}$.

The following result relates modal refinement with contract refinement as defined in the meta-theory. It justifies the consideration of modal refinement. Its proof follows by direct use of Definition 7 and Lemma 6:

Lemma 7: For $\mathcal{C}_1$ *and* $\mathcal{C}_2$ *two Modal Interfaces, the following two properties are equivalent:*

(i) $\mathcal{M}_{\mathcal{C}_2} \subseteq \mathcal{M}_{\mathcal{C}_1}$ *and* $\mathcal{E}_{\mathcal{C}_2} \supseteq \mathcal{E}_{\mathcal{C}_1}$, *meaning that contract* $\mathcal{C}_2$ *refines contract* $\mathcal{C}_1$ *following the meta-theory;*

(ii) $[\mathcal{C}_2] \preceq [\mathcal{C}_1]$, *i.e.,* $[\mathcal{C}_2]$ *refines contract* $[\mathcal{C}_1]$ *in the sense of modal refinement.*

The conjunction of two Modal Interfaces is thus the Greatest Lower Bound (GLB) with respect to refinement order. Its computation proceeds in two steps. In a first step, we wildly enforce the GLB and compute a *pre-conjunction* by taking union of must sets and intersection of may sets:

Definition 9: The pre-conjunction[43] $\mathcal{C}_1 \underline{\wedge} \mathcal{C}_2$ *of two Modal Interfaces is only defined if* $\Sigma_1^{\rm in} = \Sigma_2^{\rm in}$ *and* $\Sigma_1^{\rm out} = \Sigma_2^{\rm out}$ *and is given by* $\Sigma^{\rm in} = \Sigma_1^{\rm in}$, $\Sigma^{\rm out} = \Sigma_1^{\rm out}$, $Q = Q_1{\times}Q_2$, $q_0 = (q_{1,0}, q_{2,0})$, *and its two transition relations are given by:*

$$\begin{array}{rcl} must(q_1,q_2) & = & must_1(q_1) \cup must_2(q_2) \\ may(q_1,q_2) & = & may_1(q_1) \cap may_2(q_2) \end{array} \tag{26}$$

Due to (26), $\mathcal{C}_1 \underline{\wedge} \mathcal{C}_2$ may involve inconsistent states and, thus, in a second step, the pruning introduced in Lemma 6 must be applied:

$$\mathcal{C}_1 \wedge \mathcal{C}_2 \quad = \quad [\mathcal{C}_1 \underline{\wedge} \mathcal{C}_2] \tag{27}$$

Say that *the two Modal Interfaces* $\mathcal{C}_1$ *and* $\mathcal{C}_2$ *are consistent*[44] if $\mathcal{C}_1 \wedge \mathcal{C}_2$ is consistent in the sense of Lemma 6.

Parallel composition and quotient: For Modal Interfaces, we are able to define both parallel composition and quotient in the sense of the meta-theory. As it was the case for Interface Automata, parallel composition for Modal Interfaces raises compatibility issues and, thus, a two-step procedure is again followed for its computation.

Definition 10: The pre-composition $\mathcal{C}_1 \underline{\otimes} \mathcal{C}_2$ *of two Modal Interfaces is only defined if* $\Sigma_1^{\rm out} \cap \Sigma_2^{\rm out} = \emptyset$ *and is given by:* $\Sigma^{\rm out} = \Sigma_1^{\rm out} \cup \Sigma_2^{\rm out}$, $Q = Q_1{\times}Q_2$, $q_0 = (q_{1,0}, q_{2,0})$, *and its two transition relations are given by:*

$$\begin{array}{rcl} must(q_1,q_2) & = & must_1(q_1) \cap must_2(q_2) \\ may(q_1,q_2) & = & may_1(q_1) \cap may_2(q_2) \end{array} \tag{28}$$

Say that a state (q_1,q_2) *of* $\mathcal{C}_1 \underline{\otimes} \mathcal{C}_2$ *is* illegal *if*

$$\begin{array}{rrcl} & may_1(q_1) \cap \Sigma_2^{\rm in} & \not\subseteq & must_2(q_2) \\ \text{or} & may_2(q_2) \cap \Sigma_1^{\rm in} & \not\subseteq & must_1(q_1) \end{array}$$

Illegal states are pruned away from $\mathcal{C}_1 \underline{\otimes} \mathcal{C}_2$ *as follows. Remove, from* $\mathcal{C}_1^{may} \times \mathcal{C}_2^{may}$, *all transitions leading to* (q_1,q_2). *As performing this may create new illegal states, the same is repeated until fixpoint is reached. As a final step we delete the states that are not may-reachable. This finally yields* $\mathcal{C}_1 {\otimes} \mathcal{C}_2$, *which no more possesses illegal states.*

The above construction is justified by the following result:

Lemma 8: $\mathcal{C}_1 {\otimes} \mathcal{C}_2$ *as defined in Definition 10 instantiates the contract composition from the meta-theory.*

Proof: (sketch) $\underline{E}$ is an environment for $\underline{\mathcal{C}}$ iff for any reachable state $(q_{\underline{E}}, q_1, q_2)$ of $\underline{E} \times \mathcal{C}_1^{may} \times \mathcal{C}_2^{may}$, we have

$$rs(q_{\underline{E}}) \quad \subseteq \quad must_1(q_1) \cap must_2(q_2), \tag{29}$$

where $rs(q)$, the *ready set* of state q, is the subset of actions α such that $q \xrightarrow{\alpha}$ holds. Let M_1 be any implementation of $\mathcal{C}_1$ and consider $\underline{E}{\times}M_1$. We need to prove that $\underline{E}{\times}M_1$ is an environment for $\mathcal{C}_2$, i.e., satisfies Condition 2 of Definition 7 (we must also prove the symmetric property). Let $(q_{\underline{E}}, q_1, q_2)$ be a reachable state of $\underline{E}{\times}M_1{\times}\mathcal{C}_2^{may}$. We must prove

$$rs(q_{\underline{E}}) \cap rs_{M_1}(q_1) \cap \Sigma_2^{\rm in} \quad \subseteq \quad must_2(q_2)$$

By (29) it suffices that the following property holds:

$$rs_{M_1}(q_1) \cap \Sigma_2^{\rm in} \quad \subseteq \quad must_2(q_2) \tag{30}$$

[43] Pre-conjunction was originally denoted by the symbol & in [164], [167], [166].

[44] This is the usual wording in the literature.

However, we only know that $rs_{M_1}(q_1) \subseteq may_1(q_1)$. Hence, to guarantee (30) we must prune away illegal pairs of states. To this end, we use the same procedure as before: we make state (q_1, q_2) unreachable in $\mathcal{C}_1^{may} \times \mathcal{C}_2^{may}$ by removing all *may* transitions leading to that state. We complete the reasoning as we did for the study of consistency. □

Definition 11: The quotient $\mathcal{C}_1/\mathcal{C}_2$ *of two modal interfaces* $\mathcal{C}_1$ *and* $\mathcal{C}_2$ *is only defined if* $\Sigma_1^{\text{in}} \cap \Sigma_2^{\text{out}} = \emptyset$ *and is defined according to the following two steps procedure. First, define* $\mathcal{C}$ *as follows:* $\Sigma^{\text{out}} = \Sigma_1^{\text{out}} \setminus \Sigma_2^{\text{out}}$, $Q = Q_1 \times Q_2$, $q_0 = (q_{1,0}, q_{2,0})$, *and its two transition relations are given by:*

$$\begin{array}{rcl} must(q_1, q_2) & = & must_1(q_1) \\ may(q_1, q_2) & = & \quad [may_1(q_1) \cap \neg must_1(q_1)] \cup \\ & & \quad [must_1(q_1) \cap must_2(q_2)] \cup \\ & & \neg\, [may_1(q_1) \cup may_2(q_2)] \end{array}$$

These formulas may give raise to inconsistent states, and, thus, in a second step, we set $\mathcal{C}_1/\mathcal{C}_2 = [\mathcal{C}]$.

This definition is justified by the following result [166], showing that Definition 11 instantiates the meta-theory:

Lemma 9: $\mathcal{C}_1 \otimes \mathcal{C}_2 \preceq \mathcal{C}$ *if and only if* $\mathcal{C}_2 \preceq \mathcal{C}/\mathcal{C}_1$.

Now, it may be that $(\mathcal{C}_1/\mathcal{C}_2) \otimes \mathcal{C}_2$ possess illegal pairs of states (Definition 10), whence the following improvement, where Σ^{out} and Σ^{in} are as in Definition 11 and $may_{/}$ and $must_{/}$ refer to the quotient $\mathcal{C}_1/\mathcal{C}_2$:

Definition 12: Let $\mathcal{C}'$ *be the interface defined on the same state structure as* $\mathcal{C}_1/\mathcal{C}_2$, *with however the following modalities:*

$$\begin{aligned} may'(q_1, q_2) &= may_{/}(q_1, q_2) \cap \left(may_2(q_2) \cup \Sigma^{\text{in}}\right) \\ must'(q_1, q_2) &= must_{/}(q_1, q_2) \cup (\Sigma_1^{\text{out}} \cap \Sigma_2^{\text{out}} \cap may_2(q_2)) \end{aligned}$$

define the compatible quotient, *written* $\mathcal{C}_1/\!\!/\mathcal{C}_2$, *to be the reduction of* $\mathcal{C}'$: $\mathcal{C}_1/\!\!/\mathcal{C}_2 = [\mathcal{C}']$.

This construction is justified by the following result:

Lemma 10: The compatible quotient $\mathcal{C}_1/\!\!/\mathcal{C}_2$ *solves the following problem:*

$$\max \left\{ \mathcal{C} \;\middle|\; \begin{array}{l} \mathcal{C} \textit{ has no inconsistent state} \\ \mathcal{C} \otimes \mathcal{C}_2 \textit{ has no illegal pair of states} \\ \mathcal{C} \otimes \mathcal{C}_2 \preceq \mathcal{C}_1 \end{array} \right\}$$

Proof: Denote $\mathcal{C} = \mathcal{C}_1/\!\!/\mathcal{C}_2$ the compatible quotient defined above. The proof is threefold: (i) $\mathcal{C}$ is proved to be a solution of the inequality $\mathcal{C} \otimes \mathcal{C}_2 \preceq \mathcal{C}_1$, (ii) $\mathcal{C}$ is proved to be compatible with $\mathcal{C}_2$, and (iii) every $\mathcal{C}'$ satisfying the two conditions above is proved to be a refinement of $\mathcal{C}$.

Remark that reachable states in $(\mathcal{C}_1/\!\!/\mathcal{C}_2) \otimes \mathcal{C}_2$ are of the form (q_1, q_2, q_2) and that for every reachable state pair (q_1, q_2) in $\mathcal{C}_1/\!\!/\mathcal{C}_2$, state (q_1, q_2, q_2) is reachable in $(\mathcal{C}_1/\!\!/\mathcal{C}_2) \otimes \mathcal{C}_2$.

(i) Remark that $may' \subseteq may_{/}$ and $must' \supseteq must_{/}$. Hence, $\mathcal{C}_1/\!\!/\mathcal{C}_2 \preceq \mathcal{C}_1/\mathcal{C}_2$. Since the parallel composition is monotonic, $(\mathcal{C}_1/\!\!/\mathcal{C}_2) \otimes \mathcal{C}_2 \preceq (\mathcal{C}_1/\mathcal{C}_2) \otimes \mathcal{C}_2 \preceq \mathcal{C}_1$.

(ii) For every state (q_1, q_2, q_2), reachable in $(\mathcal{C}_1/\!\!/\mathcal{C}_2) \otimes \mathcal{C}_2$, one among the following three cases occurs:

- Assume $e \in \Sigma_1^{\text{in}}$, meaning that e is an input for both $\mathcal{C}_2$ and $\mathcal{C}_1/\!\!/\mathcal{C}_2$. Hence state (q_1, q_2, q_2) is not illegal because of e.
- Assume $e \in \Sigma_1^{\text{out}} \cap \Sigma_2^{\text{out}}$, Therefore e is an input of the compatible quotient. Remark $must'(q_1, q_2) \subseteq may_2(q_2)$. Hence, state (q_1, q_2, q_2) is not illegal because of e
- Assume $e \in \Sigma_1^{\text{out}} \cap \Sigma_2^{\text{in}}$, meaning that e is an output of the compatible quotient. Remark $may'(q_1, q_2) \subseteq must_2(q_2)$. Therefore state (q_1, q_2, q_2) is not illegal because of e.

(iii) Let $\mathcal{C}''$ be a modal interface such that $\mathcal{C}_1/\!\!/\mathcal{C}_2 \preceq \mathcal{C}'' \preceq \mathcal{C}_1/\mathcal{C}_2$. We shall prove that either $\mathcal{C}'' \preceq \mathcal{C}_1/\!\!/\mathcal{C}_2$ or that $\mathcal{C}''$ is not compatible with $\mathcal{C}_2$. Remark that every reachable state (q_1, q_2) of $\mathcal{C}_1/\!\!/\mathcal{C}_2$ is related to exactly one state q'' of $\mathcal{C}''$ by the two modal refinement relations. Assume that $\mathcal{C}''$ is not a refinement of $\mathcal{C}_1/\!\!/\mathcal{C}_2$, meaning that there exists related states (q_1, q_2) and q'' such that $may'(q_1, q_2) \subseteq may''(q'') \subseteq may_{/}(q_1, q_2)$ and $must'(q_1, q_2) \supseteq must''(q'') \supseteq must_{/}(q_1, q_2)$ and either $may'(q_1, q_2) \subsetneq may''(q'')$ or $must'(q_1, q_2) \supsetneq must''(q'')$. Remark $e \in \Sigma_1^{\text{in}}$ implies that $e \in may'(q_1, q_2)$ iff $e \in may_{/}(q_1, q_2)$ and that $e \in must'(q_1, q_2)$ iff $e \in must_{/}(q_1, q_2)$. Therefore the case $e \in \Sigma_1^{\text{in}}$ does not have to be considered.

1) Assume there exists e such that $e \in may''(q'') \setminus may'(q_1, q_2)$. Remark this implies $e \in \Sigma_1^{\text{out}} \cap \Sigma_2^{\text{in}}$, meaning that e is an output of the compatible quotient. Remark also that $e \notin must_2(q_2)$. Therefore state (q'', q_2) is illegal in $\mathcal{C}'' \otimes \mathcal{C}_2$.
2) Assume there exists e such that $e \in must'(q_1, q_2) \setminus must''(q'')$. Remark this implies $e \in \Sigma_1^{\text{out}} \cap \Sigma_2^{\text{out}}$, meaning that e is an input of the compatible quotient. Remark also that $e \in may_2(q_2)$. Therefore state (q'', q_2) is illegal in $\mathcal{C}'' \otimes \mathcal{C}_2$.

□

D. Modal Interfaces with variable alphabet

As a general principle, every relation or operator introduced in Section VIII-C (for Modal Interfaces with a fixed alphabet Σ) is extended to the case of variable alphabets by 1) extending and equalizing alphabets, and then 2) applying the relations or operators of Section VIII-C to the resulting Modal Interfaces. For all frameworks we studied so far, alphabet extension was performed using inverse projections, see Section VII-C. 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 (26) or (28) yields two different answers, namely:

$$\text{for (26)} \quad : \quad \left[\begin{array}{c} \alpha \in \mathit{may}_1(q_1) \text{ and } \alpha \in \mathit{whatever}_2(q_2) \\ \Downarrow \\ \alpha \in \mathit{whatever}(q_1, q_2) \end{array}\right]$$

$$\text{for (28)} \quad : \quad \left[\begin{array}{c} \alpha \in \mathit{must}_1(q_1) \text{ and } \alpha \in \mathit{whatever}_2(q_2) \\ \Downarrow \\ \alpha \in \mathit{whatever}(q_1, q_2) \end{array}\right]$$

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 VIII-B) nor by A/G-contracts (see the end of Section VII-C). 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 [166] for justifications). Alphabets are extended as follows, where $\Sigma' \supseteq \Sigma$. Define the *strong extension* of $\mathcal{C}$ to Σ', written $\mathcal{C}^{\uparrow\Sigma'}$, as follows:

$$\mathcal{C}^{\uparrow\Sigma'} : \left\{\begin{array}{rcl} (\Sigma^{\text{in}})^{\uparrow\Sigma'} & = & \Sigma^{\text{in}} \cup (\Sigma' \setminus \Sigma) \\ (\Sigma^{\text{out}})^{\uparrow\Sigma'} & = & \Sigma^{\text{out}} \\ Q^{\uparrow\Sigma'} & = & Q \\ q_0^{\uparrow\Sigma'} & = & q_0 \\ \mathit{may}^{\uparrow\Sigma'} & = & \mathit{may} \cup (\Sigma' \setminus \Sigma) \\ \mathit{must}^{\uparrow\Sigma'} & = & \mathit{must} \cup (\Sigma' \setminus \Sigma) \end{array}\right. \tag{31}$$

In (31), the added *may* and *must* transitions are all self-loops. The *weak extension* of $\mathcal{C}$, written $\mathcal{C}^{\Uparrow\Sigma'}$, is defined using the same formulas, with the only following modification:

$$\mathcal{C}^{\Uparrow\Sigma'} : \left\{\begin{array}{rcl} \dots & = & \dots \\ \mathit{must}^{\Uparrow\Sigma'} & = & \mathit{must} \end{array}\right. \tag{32}$$

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 VIII-C extend to variable alphabets as indicated now. In the following formulas, (Σ, Σ') is a pair such that $\Sigma \subseteq \Sigma'$, and $(\Sigma_1, \Sigma_2, \Sigma)$ denotes a triple such that $\Sigma = \Sigma_1 \cup \Sigma_2$ and satisfying the typing constraints

$$\Sigma_i^{\text{in}} = \Sigma^{\text{in}} \cap \Sigma_i \text{ and } \Sigma_i^{\text{out}} = \Sigma^{\text{out}} \cap \Sigma_i \text{ for } i = 1, 2.$$

It is understood that contract $\mathcal{C}$ has alphabet Σ, and so on.

Theorem 1: The following relations and operators

$$\begin{array}{rcl} M' \models^{\text{M}} \mathcal{C} & ::= & M' \models^{\text{M}} \mathcal{C}^{\Uparrow\Sigma'} \\ E' \models^{\text{E}} \mathcal{C} & ::= & E' \models^{\text{E}} \mathcal{C}^{\uparrow\Sigma'} \\ \mathcal{C}_1 \preceq \mathcal{C}_2 & ::= & \mathcal{C}_1 \preceq \mathcal{C}_2^{\Uparrow\Sigma} \\ \mathcal{C}_1 \wedge \mathcal{C}_2 & ::= & \mathcal{C}_1^{\Uparrow\Sigma} \wedge \mathcal{C}_2^{\Uparrow\Sigma} \\ \mathcal{C}_1 \otimes \mathcal{C}_2 & ::= & \mathcal{C}_1^{\uparrow\Sigma} \otimes \mathcal{C}_2^{\uparrow\Sigma} \\ \mathcal{C}_1 / \mathcal{C}_2 & ::= & \mathcal{C}_1^{\Uparrow\Sigma} / \mathcal{C}_2^{\uparrow\Sigma} \end{array} \tag{33}$$

instantiate the meta-theory.

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. □

E. Projecting and Restricting

A difficult step in the management of contracts was illustrated in Figure 4 of Section IV-D. It consists in decomposing a contract $\mathcal{C}$ into a composition of sub-contracts

$$\bigotimes_{i \in I} \mathcal{C}_i \quad \preceq \quad \mathcal{C} \tag{34}$$

where sub-contract $\mathcal{C}_i$ has alphabet $\Sigma_i = \Sigma_i^{\text{in}} \uplus \Sigma_i^{\text{out}}$. As a prerequisite to (34), the designer has to guess some topological architecture by decomposing the alphabet of actions of $\mathcal{C}$ as

$$\Sigma \;=\; \bigcup_{i \in I} \Sigma_i \quad , \quad \Sigma_i \;=\; \Sigma_i^{\text{in}} \uplus \Sigma_i^{\text{out}} \tag{35}$$

such that composability conditions regarding inputs and outputs hold. Guessing architectural decomposition (35) relies on the designer's understanding of the system and how it should naturally decompose—this typically is the world of SysML. Finding decomposition (34) is, however, technically difficult in that it involves behaviors. It is particularly difficult if $\mathcal{C}$ is itself a conjunction of viewpoints or requirements, which typically occurs in requirements engineering, see Section XI:

$$\mathcal{C} \;=\; \bigwedge_{k \in K} \mathcal{C}_k \tag{36}$$

The algorithmic means we develop in the remaining part of this section will be instrumental in solving (34). They will be used in the Parking Garage example of Section XI.

Projecting over a subalphabet: Projecting a contract over a sub-alphabet is the first natural tool for consideration. It is the dual operation with respect to alphabet extension that was developed in Section VIII-D. Two projections can be defined, which can be seen as under- and over-approximations, respectively, by referring to the refinement order.

Let $\mathcal{C}$ be a Modal Interface with alphabet $\Sigma = \Sigma^{\text{in}} \cup \Sigma^{\text{out}}$ and let $\Sigma' \subset \Sigma$ be a sub-alphabet. Define respectively the *abstracted projection* and *refined projection*

$$\begin{array}{lcl} \mathcal{C}^{\downarrow\Sigma'} & = & \min\{\mathcal{C}' \mid \Sigma_{\mathcal{C}'} = \Sigma' \text{ and } \mathcal{C}' \succeq \mathcal{C}\} \\ \mathcal{C}^{\Downarrow\Sigma'} & = & \max\{\mathcal{C}' \mid \Sigma_{\mathcal{C}'} = \Sigma' \text{ and } \mathcal{C}' \preceq \mathcal{C}\} \end{array} \tag{37}$$

where "min" and "max" refer to refinement order for Modal Interfaces with different alphabets, see (33). The projection $\mathcal{C}^{\Downarrow\Sigma'}$ (respectively $\mathcal{C}^{\downarrow\Sigma'}$) is computed via the following on-the-fly algorithm: 1) Perform ε-closure on the set of states by considering unobservable the actions not belonging to Σ', and then, 2) Perform determinization by giving priority to *must* over *may* (respectively to *may* over *must*), see Table VI for details. Step 2) of the procedure for computing $\mathcal{C}^{\Downarrow\Sigma'}$ is a possible source of inconsistencies.

The abstracted projection $\mathcal{C}^{\downarrow\Sigma'}$ provides a projected view of a system-level contract over a small subset of actions. The abstracted projection is thus a useful tool for contract exploration and debug. It is not a synthesis tool, however.

This is in contrast to the refined projection $\mathcal{C}^{\Downarrow\Sigma'}$, which satisfies the following result. For any finite set $\{\mathcal{C}_i \mid i \in I\}$ of Modal Interfaces with respective alphabets Σ_i, we have, assuming that the left and right hand sides are properly defined—a typing property on input and output alphabets:

$$\bigotimes_{i\in I}\left[\mathcal{C}_i \wedge \left(\bigwedge_{k\in K} \mathcal{C}_k^{\Downarrow\Sigma_i}\right)\right] \quad \preceq \quad \bigwedge_{k\in K} \mathcal{C}_k \tag{38}$$

To show this, by symmetry, it is enough to show (38) with the conjunction on the right hand side replaced by any single $\mathcal{C}_k$ for k selected from finite set K. For any $i \in I$, we have $\mathcal{C}_k^{\Downarrow\Sigma_i} \preceq \mathcal{C}_k$, and thus

$$\mathcal{C}'_i \;\; =_{\text{def}} \;\; \left[\mathcal{C}_i \wedge \left(\bigwedge_{k\in K} \mathcal{C}_k^{\Downarrow\Sigma_i}\right)\right] \preceq \mathcal{C}_k \tag{39}$$

holds for every i, from which (38) follows. Unfortunately, the refined projection quite often gives raise to inconsistencies. In addition, the contract composition arising in left hand side of (38) is subject to incompatibilities. Informally said, this technique works if the sub-systems are only "loosely coupled" so that neither inconsistencies nor incompatibilities occur. To mitigate this problem we propose an alternative technique for turning a conjunction into a composition.

Restricting to a sub-alphabet: Let $\mathcal{C}$ 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 $\Sigma'^{\text{out}} \subseteq \Sigma^{\text{out}}$. Set $\Sigma' = \Sigma'^{\text{in}} \uplus \Sigma'^{\text{out}}$ and define the *restriction* of $\mathcal{C}$ to Σ', denoted by $\mathcal{C}_{\downarrow\Sigma'}$ via the procedure shown in see Table VII, compare with Table VI. Observe that the states of the restriction correspond to sets of states of the original Modal Interface. The restriction aims at avoiding incompatibilities when considering the composition, as the following lemma shows:

Lemma 11: If $\mathcal{C}$ and $\mathcal{C}_{\downarrow\Sigma'}$ are both consistent, then so is their compatible quotient $\mathcal{C} /\!\!/ \mathcal{C}_{\downarrow\Sigma'}$, see Definition 12.

```
input: $\mathcal{C}, \Sigma, \Sigma'$; output: $\mathcal{C}'$

let proj$(Op, X)$ =
   if $X$ has not been visited,
   then
      1. mark $X$ visited
      2. for every $\alpha \in \Sigma'$ do
           2.1 let $Y = \varepsilon$-closure$(\Sigma - \Sigma', \text{next}(\alpha, X))$
           2.2 let $m = Op\{m_{\mathcal{C}}(q, \alpha) \mid q \in X\}$
           2.3 add to $\mathcal{C}'$ a transition $(X, \alpha, Y)$ with modality $m$
           2.4 proj$(Op, Y)$
         done

let project$(Op, \mathcal{C})$ =
   1. let $X_0 = \varepsilon$-closure$(\Sigma - \Sigma', q_0)$
   2. set initial state of $\mathcal{C}'$ to $X_0$
   3. proj$(Op, X_0)$
   4. reduce $\mathcal{C}'$
   5. return $\mathcal{C}'$

let order $(\{\mathit{cannot}, \mathit{may}, \mathit{must}\}, \leq) = \left\{\begin{array}{r} \mathit{must} \leq \mathit{may} \\ \mathit{cannot} \leq \mathit{may} \end{array}\right.$
   in $\mathcal{C}^{\downarrow\Sigma'}$ = project$(\vee, \mathcal{C})$
      $\mathcal{C}^{\Downarrow\Sigma'}$ = project$(\wedge, \mathcal{C})$
```

Table VI
ALGORITHM FOR COMPUTING THE PROJECTIONS (37).

```
input: $\mathcal{C}, \Sigma^{\text{in}}, \Sigma^{\text{out}}, \Sigma'^{\text{in}}, \Sigma'^{\text{out}}$; output: $\mathcal{C}'$

let order$(\{\mathit{cannot}, \mathit{may}, \mathit{must}\}, \leq^{\text{in}}) = \mathit{cannot} \leq^{\text{in}} \mathit{may} \leq^{\text{in}} \mathit{must}$
   order$(\{\mathit{cannot}, \mathit{may}, \mathit{must}\}, \leq^{\text{out}}) = \left\{\begin{array}{r} \mathit{must} \leq \mathit{may} \\ \mathit{cannot} \leq \mathit{may} \end{array}\right.$
   in let rest$(X)$ =
         if $X$ has not been visited,
         then
            1. mark $X$ visited
            2. for every $\alpha \in \Sigma'$ do
                 2.1 let $Y = \varepsilon$-closure$(\Sigma - \Sigma', \text{next}(\alpha, X))$
                 2.2 let $m = Op\{m_{\mathcal{C}}(q, \alpha) \mid q \in X\}$
                     where $Op$ = if $\alpha \in \Sigma'^{\text{in}}$ then $\vee^{\text{in}}$ else $\wedge^{\text{out}}$
                 2.3 add to $\mathcal{C}'$ a transition $(X, \alpha, Y)$ with modality $m$
                 2.4 rest$(Op, Y)$
               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$(Op, X_0)$
   4. reduce $\mathcal{C}'$
   5. return $\mathcal{C}'$
```

Table VII
ALGORITHM FOR COMPUTING THE RESTRICTION $\mathcal{C}_{\downarrow\Sigma'}$.

Proof: Consider two alphabets $\Sigma \supseteq \Sigma'$, and a consistent $\mathcal{C}$ on alphabet Σ, such that $\mathcal{C}_{\downarrow\Sigma'}$ is also consistent. The only case where quotient produces inconsistent states is whenever there exists an action e and a state pair (q, R) in $\mathcal{C} /\!\!/ \mathcal{C}_{\downarrow\Sigma'}$, such that e has modality *must* in q and does not have modality *must* in R. We prove by contradiction that no such reachable state pair (q, R) and action e exist. Remark that by definition of the restriction, $q \in R$. The restriction is assumed to be in reduced form, meaning that it does not contain inconsistent states. Two cases have to be considered:

1) Action e has modality *cannot* in R. Several sub-cases have to be considered, depending on the I/O status of e and on the fact that the reduction of the restriction has turned a *may* modality for e into a *cannot*. In all cases, action e has modality *cannot* or *may* in q, which

contradicts the assumption.

2) Action e has modality *may* in R. This implies that e also has modality *may* in q, which contradicts the assumption.

This finishes the proof of the lemma. □

The following holds by Lemma 10:

$$\begin{array}{r}\mathcal{C}_{\downarrow\Sigma'} \otimes (\mathcal{C} /\!\!/ \mathcal{C}_{\downarrow\Sigma'}) \preceq \mathcal{C} \\ \mathcal{C} /\!\!/ \mathcal{C}_{\downarrow\Sigma'} \text{ has no inconsistent state} \\ (\mathcal{C}_{\downarrow\Sigma'}, \mathcal{C} /\!\!/ \mathcal{C}_{\downarrow\Sigma'}) \text{ has no incompatible pair of states}\end{array} \tag{40}$$

By Lemma 11, the only property for checking is the consistency of $\mathcal{C}_{\downarrow\Sigma'}$. This is a significant improvement compared to the issues raised by the refined projection, as explained in the paragraph following (39). Decomposition (40) can be used while sub-contracting through the following algorithm:

Algorithm 1: We are given some system-level contract $\mathcal{C}$. The top-level designer guesses some topological architecture according to (35). Then, she recursively decomposes:

$$\begin{array}{lllllll}\mathcal{C} = \mathcal{C}_0 & \succeq & \mathcal{C}_{0\downarrow\Sigma_1} & \otimes & \mathcal{C}_1 & & \\ & \succeq & \mathcal{C}_{0\downarrow\Sigma_1} & \otimes & \mathcal{C}_{1\downarrow\Sigma_2} & \otimes & \mathcal{C}_2 \\ & \vdots & & & & & \\ & \succeq & \mathcal{C}_{0\downarrow\Sigma_1} & \otimes & \dots & \otimes & \mathcal{C}_{n-1\downarrow\Sigma_n} \\ & =_{\text{def}} & \mathcal{C}(\Sigma_1) & \otimes & \dots & \otimes & \mathcal{C}(\Sigma_n)\end{array} \tag{41}$$

which ultimately yields a refinement of $\mathcal{C}$ by a compatible composition of local sub-contracts. □

Discussion: The operations of projection and restriction that we introduced here can probably be made be generic, at the level of the meta-theory. Observe, however, that such operations explicitly refer to the notion of sub-alphabet, which does not exist at the level of the meta-theory. In addition, Modal Interfaces are the only concrete theory in which variable alphabets are properly supported. For these two reasons, we preferred to develop these operations for Modal Interfaces only.

F. Observers

Here we develop observers for a Modal Interface $\mathcal{C} = (\Sigma^{\text{in}}, \Sigma^{\text{out}}, Q, q_0, \rightarrow, \dashrightarrow)$ having no inconsistent state, meaning that $\rightarrow \subseteq \dashrightarrow$. With this consistency assumption in force, observers are then obtained as follows, with reference to Definition 2:

- Condition 2 of Definition 7 boils down to requiring that $E \times \mathcal{C}^{must}$ simulates E. Simulation testing can thus be used to check this; call $b^{\mathsf{E}}_{\mathcal{C}}(E)$ the corresponding verdict.
- To test for implementations, we first construct the maximal environment $E_{\mathcal{C}}$ and apply testing to check simulation of $E_{\mathcal{C}} \times M$ by $\mathcal{C}^{may}$, call $b^{\mathsf{M}}_{\mathcal{C},1}$ the corresponding verdict. Performing this requires maintaining pairs of states $((q_E, q_M), (q_E, q))$ in simulation relation: $(q_E, q_M) \leq (q_E, q)$. For any such pair of states, let $b^{\mathsf{M}}_{\mathcal{C},2}$ denote the verdict answering whether $(q_E, q_M) \xrightarrow{\alpha}_{E_{\mathcal{C}} \times M}$ holds each time $q \xrightarrow{\alpha}$ holds, for any $\alpha \in \Sigma^{\text{out}}$. The overall verdict for implementation testing is then

$$b^{\mathsf{M}}_{\mathcal{C},1}(E_{\mathcal{C}} \times M) \wedge b^{\mathsf{M}}_{\mathcal{C},2}(E_{\mathcal{C}} \times M)$$

Lemma 3 for generic observers specializes to the following, effective, semi-decision procedure:

Lemma 12:

1) *If* $b^{\mathsf{E}}_{\mathcal{C}}$ *outputs* F, *then* $\mathcal{C}$ *is incompatible;*
2) *If* $b^{\mathsf{M}}_{\mathcal{C},1} \wedge b^{\mathsf{M}}_{\mathcal{C},2}$ *outputs* F, *then* $\mathcal{C}$ *is inconsistent.*

G. Bibliographical note

As explained in [72], [56], [127], [83], [165], [166], Interface Theories make no explicit distinction between assumptions and guarantees.

Interface Automata, variants and extensions: Interface Automata were proposed by de Alfaro and Henzinger [72], [70], [5], [54] as a candidate theory of interfaces. In these references, Interface Automata focused primarily on parallel composition and compatibility. Quoting [72]:

> "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 [72], [56]. In contrast to our development in Section VIII-B, 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 [72]—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 [9], 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). Despite the availability of partial results [83], the framework of Interface Automata lacks support for the conjunction of interfaces in a general setting, and does not offer the flexibility of modalities. *Sociable Interfaces* [71] combine the approach presented in the previous paragraph with interface automata [72], [73] by enabling communication via shared variables and actions[45]. 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 [3]. *Software Interfaces* were proposed in [55], 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

[45] This formalism is thus not purely synchronous and is mentioned in this section with a slight abuse.

also a comprehensive interface description of Tiny OS,[46] 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* [56]. 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. In [83], the framework of *Synchronous Interfaces* is enriched with a notion of conjunction (called *shared refinement*). This development was further elaborated in [78]. In Moore interfaces, legal values of the input variables and consequent legal values of the output variables are not causally related. *Synchronous Relational Interfaces* [177], [178] 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 [112] studies conditions that need to be imposed on interface models in order to enforce independent implementability with respect to conjunction. Finally, [53] develops the concept of simulation distances for interfaces, thereby taking robustness issues into account by tolerating errors. Finally, Web services Interfaces were proposed in [38].

The discussion of variants and extensions related to time, resources, and probability, is deferred to corresponding sections.

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 [130], [11], [46] as a simple and elegant framework to express both allowed and mandatory properties. *Modal Specifications* basically consist in assigning a modality *may* or *must* to each possible transition of a system. They have been first studied in a process-algebraic context [130], [125] in order to allow for loose specifications of systems. Since then, they have been considered for automata [128] and formal languages [163], [164] and applied to a wide range of application domains (see [11] for a complete survey). Informally, a *must* transition is available in every component that realizes the modal specification, while a *may* transition needs not be. A modal specification thus represents a set of *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 VIII-B and VIII-C. Modal Specifications offer built-in conjunction of specifications [129], [167]. The expressiveness of Modal Specifications has been characterized as a strict fragment of the Hennessy-Milner logic in [46] and also as a strict fragment of the mu-calculus in [89]. The formalism is rich enough to specify safety properties as well as restricted forms of liveness properties. *Modal Interfaces* with the right notion of compatibility were introduced in [165], [166] and the problem of alphabet equalization with weak and strong alphabet extensions was first correctly addressed in the same references. In [23], 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 [23], i.e., two Modal Interfaces are only compatible if incompatibility between two interfaces can occur in any environment. A verification tool called MIO Workbench is available. The quotient of Modal Specifications was studied in [124], [164]. Determinism plays a role in the modal theory. *Non-deterministic Modal Interfaces* have possibly non-deterministic i/o-automata as class of components. They are much more difficult to study than deterministic ones and corresponding computational procedures are of higher complexity. A Modal Interface is said to be deterministic if its *may*-transition relation is deterministic. For nondeterministic Modal Interfaces, modal refinement is *incomplete* [128]: there are nondeterministic Modal Interfaces $\mathcal{C}_1$ and $\mathcal{C}_2$ for which the set of implementations of $\mathcal{C}_1$ is included in that of $\mathcal{C}_2$ without $\mathcal{C}_1$ being a modal refinement of $\mathcal{C}_2$. Hence refinement according to the meta-theory 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 EXPTIME-complete [12], [25] whereas the problem is PTIME-complete if determinism is assumed [167], [26]. The benefits of the determinism assumption in terms of complexity for various decision problems on modal specifications is underlined in [26]. The "merge" of non-deterministic Modal Specifications regarded as partial models has been considered in [179]. This operation consists in looking for common refinements of initial specifications and is thus similar to the conjunction operation presented here. In [179], [90], algorithms to compute the maximal common refinements (which are not unique when non-determinism is

[46] http://www.tinyos.net/

allowed) are proposed. They are implemented in the tool MTSA [82]. Assume/guarantee contracts viewed as pairs of Modal Specifications were proposed in [96]. 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 [173], Existential Live Sequence Charts are translated into Modal Specifications, hence providing a mean to specify modal contracts. Regarding extensions, *Acceptance Interfaces* were proposed by J-B. Raclet [163], [164]. 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. In intuitive terms, an Acceptance Interface 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 [27]. 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 [4] to the separate compilation of multiple clocked synchronous programs.

The discussion of variants and extensions related to time, resources, and probability, is deferred to corresponding sections.

Features of our presentation: The presentation of interface theories in this paper 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 $\mathcal{E}_c$ and $\mathcal{M}_c$ 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 projection and restriction operators for Modal Interfaces are new. 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 [49], which implements Modal Interfaces with all the operations and services discussed in this section.

IX. Panorama: Timed Interface Theories

In this section we develop an instance of an interface theory for timed systems. This section is meant to be illustrative of our approach to concrete instances of contract theories. It does not claim to expose *the* good instance. Our presentation builds on top of [35]. To simplify the exposure, we restrict ourselves to the case of a fixed alphabet Σ. As usual now, we begin with the component model.

A. Components as Event-Clock Automata

Timed Automata [6] 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 [7], [8], [35] form a subclass of timed automata where clock resets are not arbitrary: each action α comes with a clock h_α which is reset exactly when action α occurs. The interest of this subclass is that event-clock automata are determinizable, which facilitates the development of a (modal) theory of contracts. The definition of event-clock automata requires some prerequisites.

We are given an underlying finite set $\mathcal{H}$ of *clocks*. A *clock valuation* is a map $\nu : \mathcal{H} \mapsto \mathbb{R}_+$, where $\mathbb{R}_+$ denotes the set of nonnegative reals. The set of all clock valuations is denoted by $\mathcal{V}$. We denote by $\mathbf{0}$ the clock valuation assigning 0 to all clocks and by $\nu + t$ the clock obtained by augmenting ν with the constant t. For ν a valuation, the valuation $\nu[\mathbf{0}/h_\alpha]$ assigns 0 to h_α and keeps the valuation of other clocks unchanged.

Clocks can be used to define guards. The class of guards considered by [35] consists of finite conjunctions of expressions of the form $h \sim n$ where h is a clock, n is an integer, and $\sim$ is one of the following relations: $<, \leq, =, \geq, >$. For g a guard, write $\nu \models g$ to mean that valuation ν satisfies guard g. For N an integer, $\mathcal{G}(\mathcal{H}, N)$ denotes the set of all guards over $\mathcal{H}$ involving only integers $n < N + 1$ and we write $\mathcal{G}(\mathcal{H}) = \mathcal{G}(\mathcal{H}, \infty)$.

Definition 13: An event-clock automaton *over alphabet* Σ *is a tuple* $M = (\Sigma^{\text{in}}, \Sigma^{\text{out}}, Q, q_0, N, \rightarrow)$*, where:*

- $\Sigma^{\text{in}}, \Sigma^{\text{out}}, Q, q_0$ *are as for i/o-automata; elements of* Q *are called* locations*;*
- $\mathcal{H}_\Sigma = \{h_\alpha \mid \alpha \in \Sigma\}$ *is the set of* clocks*;*
- N *is a finite integer and* $\rightarrow \subseteq Q \times \mathcal{G}(\mathcal{H}_\Sigma, N) \times \Sigma \times Q$ *is the* transition relation*; write* $q \xrightarrow{g,\alpha} q'$ *to mean that* $(q, g, \alpha, q') \in \rightarrow$.

The semantics of M is a timed transition system with infinite state space $Q \times \mathcal{V}$, defined as follows: transition $q \xrightarrow{g,\alpha} q'$ can fire from state $(q, \nu) \in Q \times \mathcal{V}$ iff $\nu \models g$; as a result of this firing, the clock h_α is reset to zero, thus resulting in the new state $(q', \nu[\mathbf{0}/h_\alpha]) \in Q \times \mathcal{V}$. In our Timed Interface theory, *components* are event-clock automata that are:

1) *deterministic* in the following sense:

$$\begin{array}{l}\forall (q, \alpha, \nu) \in Q \times \Sigma \times \mathcal{V}, \text{ there is at most} \\ \text{one transition } q \xrightarrow{g,\alpha} q' \text{ such that } \nu \models g.\end{array} \quad (42)$$

2) *receptive*, meaning that

$$\forall q \in Q, \forall \alpha \in \Sigma^{\text{in}} : q \xrightarrow{\text{T},\alpha}$$

holds, where T denotes the trivial guard "true".

Instead of further developing our theory we will show how to take advantage of previously developed theories by showing that event-clock automata are finitely encoded by means of *region automata* that we introduce next. Fix bound N. A *region* is an equivalence class ϑ of clock valuations ν satisfying the same set of guards from $\mathcal{G}(\mathcal{H}, N)$. Denote by Θ the set of all such regions (bound N is understood). For $\vartheta \in \Theta$, consider

$$\tau(\vartheta) \quad = \quad \{\vartheta'' \mid \exists \nu'' \in \vartheta'', \exists \nu \in \vartheta, \exists t \geq 0 \text{ s.t. } \nu'' = \nu + t\}$$

which is the set of all regions that can be obtained from ϑ by letting time elapse. Finally, for ϑ a region and α an action, $\vartheta_{\downarrow\alpha}$ is the region obtained from ϑ by resetting clock h_α. Using these notations, we associate, to each event-clock automaton $M = (\Sigma^{\text{in}}, \Sigma^{\text{out}}, Q, q_0, N, \rightarrow)$, a *Region Automaton*, which is the following finite state automaton:

$$\mathbb{R}\left[M\right] = \left(\Sigma^{\text{in}} \times \Theta, \Sigma^{\text{out}} \times \Theta, Q \times \Theta, (q_0, \mathbf{0}), \Rightarrow\right) \tag{43}$$

where the transition relation $\Rightarrow$ is given by:

$$(q, \vartheta) \xrightarrow{\vartheta\text{''}, \alpha} \!\!\!\!\Rightarrow (q', \vartheta') \text{ iff} \left\{ \begin{array}{l} q \xrightarrow{g, \alpha} q', \text{ with} \\ \vartheta\text{''} \subseteq \tau(\vartheta) \cap g \text{ and} \\ \vartheta' = \vartheta\text{''}_{\downarrow\alpha} \end{array} \right. \tag{44}$$

Vice-versa, any Region Automaton R defines a unique event-clock automaton $\mathbb{M}\left[R\right]$ by deducing, from (44), the minimal guards associated to its transitions. Starting from an event-clock automaton M, $\mathbb{M}\left[\mathbb{R}\left[M\right]\right]$ is related to M by a strengthening of its guards. However, it holds that[47]

$$\mathbb{R}\left[\mathbb{M}\left[\mathbb{R}\left[M\right]\right]\right] \quad = \quad \mathbb{R}\left[M\right] \tag{45}$$

Under the above correspondence, components in the event-clock automaton domain are mapped to components in the i/o-automaton domain.

B. Modal Event-Clock Specifications

In this section we develop the framework of contracts on top of our component model of deterministic event-clock automata. To simplify, we only develop the case of a fixed alphabet Σ.

Definition 14: A Modal Event-Clock Specification *(*MECS*) is a tuple* $\mathcal{C} = (\Sigma^{\text{in}}, \Sigma^{\text{out}}, Q, q_0, \rightarrow, \dashrightarrow)$*, where* $\Sigma^{\text{in}}, \Sigma^{\text{out}}$*,* Q, q_0 *are as in Interface Automata and*

$$\rightarrow \;\subseteq\; \dashrightarrow \;\subseteq\; Q \times \mathcal{G}(\mathcal{H}_\Sigma) \times \Sigma \times Q$$

Using the same construction as above, MECS $\mathcal{C}$ *induces a* Region Modal Event-Clock Specification *(*RMECS*) denoted by* $\mathbb{R}\left[\mathcal{C}\right]$*, which is a Modal Interface according to Definition 7. Accordingly,* $\mathbb{R}\left[\mathcal{C}\right]$ *defines a pair* $(\mathcal{E}_{\mathbb{R}[\mathcal{C}]}, \mathcal{M}_{\mathbb{R}[\mathcal{C}]})$ *of sets of i/o-automata and we finally define*

$$(\mathcal{E}_{\mathcal{C}}, \mathcal{M}_{\mathcal{C}}) \quad =_{\text{def}} \quad \left(\mathbb{M}\left[\mathcal{E}_{\mathbb{R}[\mathcal{C}]}\right], \mathbb{M}\left[\mathcal{M}_{\mathbb{R}[\mathcal{C}]}\right]\right)$$

The map $\mathcal{C} \mapsto \mathbb{R}\left[\mathcal{C}\right]$ satisfies:[48]

$$\mathbb{R}\left[\mathbb{M}\left[\mathbb{R}\left[\mathcal{C}\right]\right]\right] \quad = \quad \mathbb{R}\left[\mathcal{C}\right] \tag{46}$$

This allows us to lift the contract theory of RMECS (which are Modal Interfaces) to a contract theory of MECS. While this is conceptually elegant, it is not efficient since moving from MECS to RMECS (from timed models to region models) is ExpTime, which is costly. To cope with this difficulty, the authors of [35] have provided direct cheap formulas, expressed in the MECS domain. These formulas provide a partial answer to the issue of complexity. In particular, the following lemma holds that was stated and proved in [35]:

Lemma 13: For $\mathcal{C}_1$ *and* $\mathcal{C}_2$ *two* MECS *having identical input and output alphabets, define* $\mathcal{C}_1 \underline{\triangle} \mathcal{C}_2$ *symbolically with rules of the following form:*

$$\text{Glb}_{may} \quad : \quad \frac{q_1 \overset{g_1, \alpha}{\dashrightarrow} q_1' \textit{ and } q_2 \overset{g_2, \alpha}{\dashrightarrow} q_2'}{(q_1, q_2) \overset{g_1 \wedge g_2 . \alpha}{\dashrightarrow} (q_1', q_2')}$$

$$\text{Glb}_{must,\text{left}} \quad : \quad \frac{q_1 \overset{g_1, \alpha}{\rightarrow} q_1' \textit{ and } q_2 \overset{g_2, \alpha}{\rightsquigarrow} q_2'}{(q_1, q_2) \overset{g_1 \wedge g_2 . \alpha}{\rightarrow} (q_1', q_2')}$$

where $\rightsquigarrow$ *denotes ad libitum* $\rightarrow$ *or* $\dashrightarrow$ *and with* $\text{Glb}_{must,\text{right}}$ *defined symmetrically (see [35] for the complete set of rules). Then, we have*

$$\mathbb{R}\left[\mathcal{C}_1 \underline{\triangle} \mathcal{C}_2\right] \quad = \quad \mathbb{R}\left[\mathcal{C}_1\right] \underline{\triangle} \mathbb{R}\left[\mathcal{C}_2\right] \tag{47}$$

Inconsistent states for both sides of (47) correspond. A similar lemma holds for contract quotient $\mathcal{C}_1/\mathcal{C}_2$. Unfortunately, the pruning of inconsistent states cannot be performed directly in the MECS domain and mapping to regions is mandatory. Nevertheless, dedicated rules to handle directly inconsistent states exist in the symbolic versions of the different relations and operations on MECS. As a result, the computation of the regions is only mandatory when the designer aims at cleaning its contract by pruning the inconsistencies. A similar lemma holds for contract composition $\mathcal{C}_1 \otimes \mathcal{C}_2$. Again, pruning incompatible pairs of states cannot be performed directly in the MECS domain and mapping to regions is mandatory.[49]

C. Bibliographical note

This note focuses on interface types of framework. The reader is referrect to the bibliographical note of Section VII-G for extensions of A/G-contracts addressing time.

A first interface theory able to capture the timing aspects of components is *Timed Interfaces* [74]. 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 [68] fills a gap in the work introduced in [74] 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

[47] It is shown in [35] that $M \mapsto \mathbb{R}\left[M\right]$ forms a Galois connection.

[48] Again, it is a Galois connection [35].

[49] Authors of [35] do not mention this fact for the following reason. They do not consider the issue of receptiveness for components and therefore the issue of compatibility does not arise.

implemented in the tool ECDAR [69]. Timed Specification Theories are revisited from a linear-time perspective in [60].

The first *timed* extension of modal transition systems was published in [52]. 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 [95]. Another timed extension of Modal Specifications was proposed in [36]. 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 [33], [34].

Resources other than time were also considered—with energy as the main target. *Resource Interfaces* [57] can be used to enrich a variety of interface formalisms (Interface Automata [72], Assume/Guarantee Interfaces [73], 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 then 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.

X. PANORAMA: PROBABILISTIC INTERFACE THEORIES

Probabilistic systems are non-deterministic systems in which choice is controlled by random trial. Such systems are useful for many purposes, ranging from physical or biological systems, to security protocols, intrusion detection, and up to safety and reliability analyses in system design. In this section, as another illustration of our meta-theory, we develop a simple instance of a probabilistic interface theory. Again, to simplify the exposure, we restrict ourselves to the case of a fixed alphabet Σ.

A. Components as Probabilistic Automata

We first introduce *Probabilistic Automata* with inputs and outputs, also called *Input/Output Markov Decision Processes*, and then we motivate their use in safety analysis.

Definition 15: A Probabilistic Automaton *with inputs and outputs (i/o-PA) is a tuple* $M = (\Sigma^{\text{in}}, \Sigma^{\text{out}}, Q, q_0, \Pi, \rightarrow)$, *where*

- Σ^{in} *and* Σ^{out} *are disjoint finite input and output alphabets such that* $\Sigma^{\text{in}} \cup \Sigma^{\text{out}} = \Sigma$*;*
- Q *is a finite set of* states *and* $q_0 \in Q$ *is the* initial state*; we denote by* $\Pi(Q)$ *the finite set of all probabilities over* Q*, also called the* probabilistic states*;*
- $\rightarrow \subseteq Q \times \Sigma \times \Pi(Q)$ *is the* transition relation*; we write* $q \xrightarrow{\alpha} \pi$ *to mean* $(q, \alpha, \pi) \in \rightarrow$*, and* $q \xrightarrow{\alpha}$ *if* $q \xrightarrow{\alpha} \pi$ *holds for some* π*;*
- *In addition, we require* M *to be* deterministic *in the following sense: for any pair* $(q, \alpha) \in Q \times \Sigma$*,* $q \xrightarrow{\alpha} \pi$ *and* $q \xrightarrow{\alpha} \pi'$ *implies* $\pi = \pi'$*.*

A run σ of M starts from initial state q_0 and then progresses by a sequence of steps of the form $q \xrightarrow{\alpha} \pi \mapsto q'$, where supplementary transitions $\pi \mapsto q'$ are drawn at random by selecting a successor state $q' \in Q$ according to probability π. A *strategy* consists in fixing a function $\varphi : Q \rightarrow \Sigma$ and using it to select the next action. Once a strategy has been fixed, jumping from a probabilistic state to the next one gives raise to a time-invariant Markov Chain, whose invariant probability can be computed or statistically estimated. When all probabilistic states are Dirac measures $\delta_{q'}$ selecting q' as a next state with probability one, i/o-PA M boils down to an i/o-automaton. Thus, i/o-PA subsume i/o-automata.

Two i/o-PA M_1 and M_2 are *composable* if $\Sigma_1^{\text{out}} \cap \Sigma_2^{\text{out}} = \emptyset$. Two composable i/o-PA M_1 and M_2 compose into $M = M_1 \times M_2$ as follows:

$$(q_1, q_2) \xrightarrow{\alpha}_M \pi_1 \otimes \pi_2 \quad \text{iff} \quad \left\{ \begin{array}{l} q_1 \xrightarrow{\alpha}_{M_1} \pi_1 \text{ and} \\ q_2 \xrightarrow{\alpha}_{M_2} \pi_2 \end{array} \right. \tag{48}$$

where $\otimes$ denotes the product of two probabilities, thus making the two components independent. Parallel composition defined in this way has all the desired properties.

Similarly, the concept of *simulation relation* is easily derived from that of i/o-automata by preserving probabilities: for $M_i, i = 1, 2$ two i/o-PA and two states $q_i \in Q_i$, say that q_1 *simulates* q_2, written $q_2 \leq q_1$ if there exists a surjective map $f : Q_1 \mapsto Q_2$ such that:

$$\begin{array}{c} \forall \alpha \text{ s.t. } q_2 \xrightarrow{\alpha}_2 \pi_2, \; \exists \pi_1 \in \Pi(Q_1) \\ \Downarrow \\ \left[\begin{array}{c} q_1 \xrightarrow{\alpha}_1 \pi_1 \\ \pi_2(q'_2) = \sum_{f(q'_1)=q'_2} \pi_1(q'_1) \\ \pi_1(q'_1) > 0 \Rightarrow f(q'_1) \leq q'_1 \end{array} \right] \end{array} \tag{49}$$

Observe that, since we consider only deterministic i/o-PA, there exists at most one π_2 such that $q_2 \xrightarrow{\alpha}_2 \pi_2$, hence there is no need to quantify over π_2 in the precondition of (49). In probability theory, relation $\pi_2(q'_2) = \sum_{f(q'_1)=q'_2} \pi_1(q'_1)$ writes $\pi_2 = f(\pi_1)$. Say that M_1 *simulates* M_2, written $M_2 \leq M_1$, if $q_{2,0} \leq q_{1,0}$. Simulation relation is preserved by composition. If $M_2 \leq M_1$, then every strategy $\varphi_2 : Q_2 \mapsto \Sigma_2$ defined over M_2 has its counterpart into a *matching* strategy $\varphi_1 : Q_1 \mapsto \Sigma_1$ over M_1 by setting $\varphi_1(q_1) = \varphi_2(f(q_1))$. For $i = 1, 2$, let $\Omega_i = Q_i^{\mathbb{N}}$ be the set of all sequences of states of Q_i, and let $(\Omega_i, \mathbb{P}_i), i = 1, 2$ be the two resulting probability spaces when matching strategies are used for M_1 and M_2. Then, the map f induces a measurable map $\psi = (f, f, \dots) : \Omega_1 \mapsto \Omega_2$ such that $\int Z(\omega_2) d\mathbb{P}_2(\omega_2) = \int Z(\psi(\omega_1)) d\mathbb{P}_1(\omega_1)$. In particular, the probability distributions with respect to $\mathbb{P}_1$ and $\mathbb{P}_2$, of any measurable function depending only on the successive actions performed, are identical.

The natural use of i/o-PAs for safety/reliability analysis is to capture faulty situations through states. Component states are partitioned into safe states and faulty states that can be equipped with different fault labels (names of states can do this, no semantic adaptation is needed). The probabilistic part of i/o-PAs naturally captures the spontaneous occurrence of faults. When in a faulty state, a component outputs actions that can either be used to model fault propagation, or may be alarms for use in system supervision. Input actions in a faulty state can be used to either model denial of service from other components (thus resulting in a risk of moving to a faulty state), or may be reconfiguration actions. Thus, spontaneous occurrence of failures with fault propagation is captured by this framework. To summarize, i/o-PAs offer a natural framework in which to combine functional and safety viewpoints.

B. Simple Modal Probabilistic Interfaces

By following the same steps as for Modal Interfaces, see Definition 7, we can define *Simple Modal Probabilistic Interfaces* as pairs $\mathcal{C} = (\mathcal{C}^{may}, \mathcal{C}^{must})$ of i/o-PAs. There is no new difficulty in doing this. Expressiveness of this framework, however, is modest; whence the attribute "simple". When used in the context of safety/reliability analysis, Simple Modal Probabilistic Interfaces support the specification of fault accommodation in systems. For example, one can specify about a component:

- that it does not accommodate denial of service: inputs representing fault propagation are disallowed ("must not occur");
- that it must be prepared to possible denial of service: inputs representing fault propagation are allowed ("must be accepted");
- that it never generates spontaneous failures: actions leading to a probabilistic state from which a faulty state can be immediately reached are disallowed ("must not occur");
- that spontaneous failures are unavoidable: actions leading to a probabilistic state from which a faulty state can be immediately reached are allowed ("must be accepted").

One could argue that i/o-PA by themselves offer a similar expressiveness by playing with probabilities. Simple Modal Probabilistic Interfaces, however, offer the full algebra of contracts, which i/o-PA don't.

C. Bibliographical note

This note focuses on interface types of framework. The reader is referred to the bibliographical note of Section VII-G for extensions of A/G-contracts addressing probability.

Exactly like the Interval Markov Chain (IMC) formalism [116] they generalize, *Constraint Markov Chains* (CMC) [50] 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 labelling 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. *Abstract Probabilistic Automata* (APA) [79] 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. For both formalisms, parallel composition is restricted to non-interacting components, since alphabets of actions must be disjoint.

Our formalism of Simple Modal Probabilistic Interfaces exhibits the whole contract algebra. Regarding limitations, unlike IMC, CMC, and APA, this model does not allow for specifying families of transition probabilities and, hence, it cannot help for quantitative reasoning about reliability analysis.

XI. The Parking Garage, an example in Requirements Engineering

In this section we develop the parking garage example introduced in Section IV-C. Some of the material is repeated for better readability. Despite being simple and small, this example quickly becomes complex for reasons that are intrinsic to the formal management of requirements. The MICA tool developed by Benoît Caillaud was used to develop it [49].

Requirements are written in constrained English language and then translated into Modal Interfaces—while the presented translation is manual, automatic translation can be envisioned.[50] 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 subsystems, each one equipped with its own requirements. The latter can then be submitted to independent suppliers for further development.

A. The contract framework

We shall use Modal Interfaces (with variable alphabet) as developed in Sections VIII-C and subsequent ones. There are three main reasons for this 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 important for requirements engineering.

[50] In fact, the contract specification languages proposed in the projects SPEEDS [29] and CESAR (http://www.cesarproject.eu/) are examples of translations from a constrained English language to a formal models of contracts similar to Modal Interfaces.

2) Being large sets of requirements structured into chapters, requirements documents are a very fragmented style of specification. Only 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) As we shall see, 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. Only Modal Interfaces provide significant assistance for this.

Overall, the problem considered in 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

$$\Sigma \;=\; \bigcup_{i\in I}\Sigma_i \quad , \quad \Sigma_i \;=\; \Sigma_i^{\rm in} \uplus \Sigma_i^{\rm out} \tag{50}$$

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

$$\bigotimes_{i\in I}\mathcal{C}_i \;\preceq\; \mathcal{C} \tag{51}$$

where sub-contract $\mathcal{C}_i$ has alphabet $\Sigma_i = \Sigma_i^{\rm in} \uplus \Sigma_i^{\rm out}$. Guessing architectural decomposition (50) relies on the designer's understanding of the system and how it should naturally decompose—this typically is the world of SysML. Finding decomposition (51) is, however, technically difficult in that it involves behaviors. The algorithmic means that were presented in Section VIII-E provide the due answer. In this Parking Garage example, we will use both the abstracted projection and the restriction that were developed in that section.

B. Top level requirements

In this section, we begin with the top-level requirements. The system under specification is a parking garage subject to payment by the user. At its most abstract level, the requirements document comprises the different chapters **gate**, **payment**, and **supervisor**, see Table VIII. The **gate** chapter will collect the generic requirements regarding entry and exit gates. These generic requirements will then be specialized for entry and exit gates, respectively.

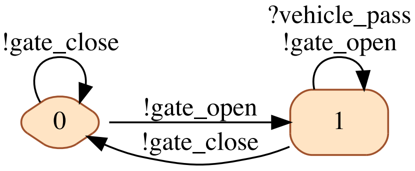

Figure 7. Requirement $\mathrm{R}_{\mathrm{g}.1}$ specified as an i/o-automaton. Prefix "?" indicates an input and prefix "!" indicates an output.

gate	
$\mathrm{R}_{\mathrm{g}.1}$:	*"vehicles shall not pass when gate is closed", see Fig. 7*
$\mathrm{R}_{\mathrm{g}.2}$:	*after ?vehicle_pass ?vehicle_pass is forbidden*
$\mathrm{R}_{\mathrm{g}.3}$:	after !gate_open !gate_open is forbidden and after !gate_close !gate_close is forbidden
payment	
supervisor	

Table VIII
THE TOP-LEVEL SPECIFICATION, WITH CHAPTER **gate** EXPANDED; REQUIREMENTS WRITTEN IN *italics* ARE ASSUMPTIONS UNDER WHICH **gate** SHOULD OPERATE.

Focus on the "**gate**" chapter. It consists of the three requirements shown on Table VIII. Requirement $\mathrm{R}_{\mathrm{g}.1}$ is best described by means of an i/o-automaton, shown in Figure 7—we provide an informal textual explanation for it, between quotes.[51] 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 $\mathrm{R}_{\mathrm{g}.1}$ and $\mathrm{R}_{\mathrm{g}.2}$, and 2) we should handle them differently than $\mathrm{R}_{\mathrm{g}.3}$, which is a guarantee offered by the system. Indeed, $\mathrm{R}_{\mathrm{g}.1}$ and $\mathrm{R}_{\mathrm{g}.2}$ are assumptions under which the gate operates as guaranteed. In the sequel, assumptions are written in *italics*.

So far we have specified **gate** as a list of requirements. Requirement $\mathrm{R}_{\mathrm{g}.1}$ specified as an i/o-automaton can be considered formal. Requirements $\mathrm{R}_{\mathrm{g}.2}$ and $\mathrm{R}_{\mathrm{g}.3}$ are formulated in constrained natural language and are ready for subsequent formalization, e.g., as i/o-automata. Are we done? Not yet! We need to give a formal meaning to what it means to have a collection of requirements, and what it means to distinguish assumptions from guarantees. The contract framework we develop next will provide support for this.

C. Formalizing requirements as contracts

We shall use Modal Interfaces (with variable alphabet) as developed in Sections VIII-C and subsequent ones. Thus, we first need to explain how the specification of "**gate**" in Table VIII translates into Modal Interfaces.

We first observe that each requirement $\mathrm{R}_{\mathrm{g.j}}$ of Table VIII is a sentence that can be formalized as an i/o-automaton, see

[51] We take the convention that the variables of the resulting i/o-automaton are those mentioned in the text of the requirement. Some adjustment is needed to this convention, however, as exemplified by Figure 7. Suppose that some requirement says: "?gate_open never occurs". This is translated by having no mention of ?gate_open in the corresponding i/o-automaton. To express this "negative" fact we must keep track of the fact that ?gate_open belongs to the alphabet of actions of the i/o-automaton. Thus, when performing the translation, the explicit list of inputs and outputs should be explicitly given. To avoid such additional notational burden, we have cheated by omitting this unless necessary.

Figure 7 for such a formalization of requirement $\mathrm{R}_{\mathrm{g}.1}$. Accordingly, each chapter $\mathrm{D} = \mathbf{gate}/\mathbf{payment}/\mathbf{supervisor}$ of Table VIII is structured as a pair

$$\mathrm{D} \quad = \quad ((A_1, \ldots, A_m), (G_1, \ldots, G_n)) \tag{52}$$

where the A_is (the assumptions) and the G_js (the guarantees) are i/o-automata in which all involved actions possess the same status (input or output) throughout all A_is and G_js. The translation of a specification of the form (52) into a Modal Interface is performed by applying the following two series of rules:

Rules 1: We adopt the following rules for the translation of guarantees (in the form of i/o-automata) into Modal Interfaces:

$\mathbf{R}^{\mathrm{G}}_1$: Unless otherwise explicitly stated, transitions labeled by an output action of the considered system are given a *may* modality; the rationale for doing this is that the default semantics for guarantees is “best effort”. A *must* modality is assigned if the requirement specifies it—e.g., by having a “must” in the sentence.

$\mathbf{R}^{\mathrm{G}}_2$: Transitions labeled by an input action of the considered system are given a *must* modality; the rationale for doing this is that, as part of its guarantees, the component must not refuse an input that is “legally” submitted by the environment.

$\mathbf{R}^{\mathrm{G}}_3$: Guarantees in a same requirements chapter D combine by conjunction.

Applying Rules 1 to the set of guarantees of D yields the contract encoding these guarantees, denoted by G_{D}. □

Performing this for the single guarantee $\mathrm{R}_{\mathrm{g}.3}$ of **gate** yields the Modal Interface shown in Figure 8.

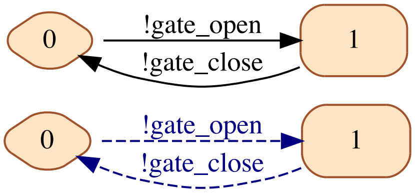

Figure 8. Translating the guarantee $\mathrm{R}_{\mathrm{g}.3}$ of **gate** as an i/o-automaton (top) and then as a Modal Interface $G_{\mathbf{gate}}$ (bottom) using Rules 1.

Rules 2: We adopt the following rules for the translation of assumptions (in the form of i/o-automata) into Modal Interfaces:

$\mathbf{R}^{\mathrm{A}}_1$: We exchange the status input/output in every assumption, thus taking the point of view of the environment.

$\mathbf{R}^{\mathrm{A}}_2$: Having done this, we apply Rules 1 to the assumptions. So far this yields a draft Modal Interface $\widehat{A}$ that must be satisfied by every environment.

$\mathbf{R}^{\mathrm{A}}_3$: This is not enough, however, as we really want environments that submit all legal stimuli. To ensure this, we simply turn, in $\widehat{A}$, all *may* transitions to *must* transitions, which yields A.

Applying Rules 2 to the set of assumptions of D yields the Modal Interface A_{D} encoding these assumptions. □

Performing this for the assumptions $\mathrm{R}_{\mathrm{g}.1}$ and $\mathrm{R}_{\mathrm{g}.2}$ of **gate** yields the Modal Interface shown in Figure 9.

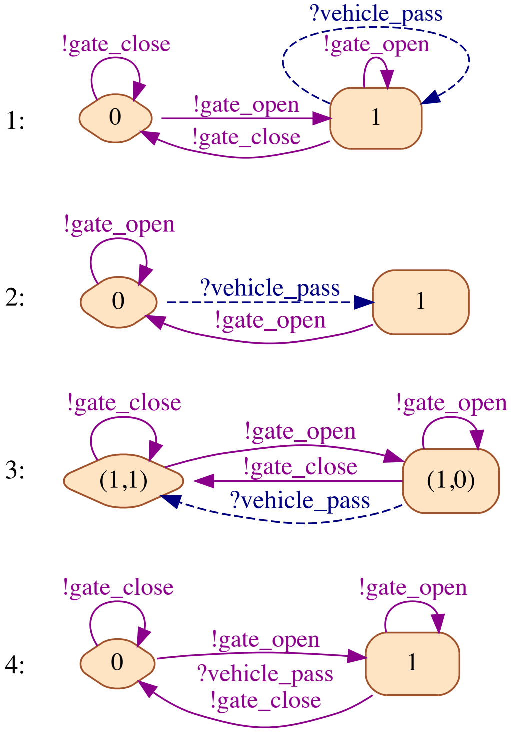

Figure 9. Translating the assumptions of **gate** as a Modal Interface $A_{\mathbf{gate}}$ using Rules 2. 1: translation of $\mathrm{R}_{\mathrm{g}.1}$; 2: translation of $\mathrm{R}_{\mathrm{g}.2}$; 3: the resulting conjunction; 4: the final result after applying rule $\mathbf{R}^{\mathrm{A}}_3$ and renaming the states.

So far we have represented any chapter D of the requirements document as a pair of Modal Interfaces $(A_{\mathrm{D}}, G_{\mathrm{D}})$ with mirroring input/output statuses for their actions—every input of A_{D} is an output of G_{D} and vice-versa. It is the merit of Modal Interfaces to allow for a direct representation of this pair of assumption and guarantee in the form of a single Modal Interface, by using the following formula, for A and G two Modal Interfaces with mirroring input/output statuses for their actions:

$$(A, G) \text{ is represented by the quotient } (A \otimes G)/A\,, \tag{53}$$

which is the contract characterizing the components that implement G in the context of A, see Section V-E. Performing this for the whole chapter **gate** yields the Modal Interface shown in Figure 10. Some comments are in order regarding this Modal Interface:

- *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_through when exiting state “1”) have a *must* modality. This complies with the intuition that

$\mathbf{gate}(x)$ where $x \in$ {entry, exit}

$\mathrm{R}_{\mathrm{g}.1}(x)$: *“vehicles shall not pass when x_gate is closed”, see Fig. 7*

$\mathrm{R}_{\mathrm{g}.2}(x)$: *after ?vehicle_pass ?vehicle_pass is forbidden*

$\mathrm{R}_{\mathrm{g}.3}$: after !x_gate_open !x_gate_open is forbidden and after !x_gate_close !x_gate_close is forbidden

$\mathbf{payment}$

$\mathrm{R}_{\mathrm{p}.1}$: *“user inserts a coin every time a ticket is inserted and only then”, Fig. omitted*

$\mathrm{R}_{\mathrm{p}.2}$: *“user may insert a ticket only initially or after an exit ticket has been issued”, Fig. omitted*

$\mathrm{R}_{\mathrm{p}.3}$: “exit ticket is issued after ticket is inserted and payment is made and only then”, Fig. omitted

$\mathbf{supervisor}$

$\mathrm{R}_{\mathrm{g}.1}$(entry)

$\mathrm{R}_{\mathrm{g}.1}$(exit)

$\mathrm{R}_{\mathrm{g}.2}$(entry)

$\mathrm{R}_{\mathrm{g}.2}$(exit)

$\mathrm{R}_{\mathrm{s}.1}$: initially and after !entry_gate close !entry_gate open is forbidden

$\mathrm{R}_{\mathrm{s}.2}$: after !ticket_issued !entry_gate open must be enabled

$\mathrm{R}_{\mathrm{s}.3}$: “at most one ticket is issued per vehicle entering the parking and tickets can be issued only if requested and ticket is issued only if the parking is not full”, see Fig 12

$\mathrm{R}_{\mathrm{s}.4}$: “when the entry gate is closed, the entry gate may not open unless a ticket has been issued”, Fig. omitted

$\mathrm{R}_{\mathrm{s}.5}$: “the entry gate must open when a ticket is issued”, Fig. omitted

$\mathrm{R}_{\mathrm{s}.6}$: “exit gate must open after an exit ticket is inserted and only then”, Fig. omitted

$\mathrm{R}_{\mathrm{s}.7}$: “exit gate closes only after vehicle has exited parking”, Fig. omitted

Table IX
REQUIREMENTS FOR THE TOP-LEVEL

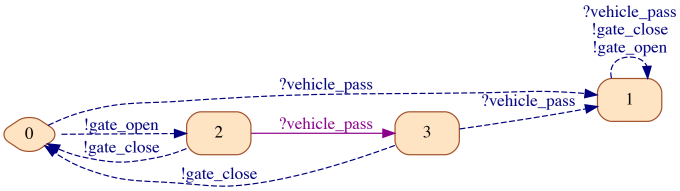

Figure 10. Chapter $\mathbf{gate}$ of the top-level requirements document translated into a Modal Interface $\mathcal{C}_{\mathbf{gate}}$.

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). As a consequence, the whole contract gets relaxed, which is reflected by the move to the special state “2”, from which any action is allowed—such a state is often called a “top” state. Note that this top state resulted from computing the quotient.

The same procedure applies to all chapters $\mathbf{gate}$, $\mathbf{payment}$, and $\mathbf{supervisor}$, of the top-level textual specification, shown in Table IX (it is an expansion of Table VIII). In particular, the Modal Interfaces encoding chapters $\mathbf{payment}$ and $\mathbf{supervisor}$ of the top-level are displayed in Figures 11 to 13. Figure 13 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 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 $\mathcal{C}$ is the conjunction of the contracts representing chapters $\mathbf{gate}$, $\mathbf{payment}$, and $\mathbf{supervisor}$, of the top-level requirements document:

$$\mathcal{C} \quad = \quad \mathcal{C}_{\mathbf{gate}} \wedge \mathcal{C}_{\mathbf{payment}} \wedge \mathcal{C}_{\mathbf{supervisor}} \tag{54}$$

Owing to the complexity of $\mathcal{C}_{\mathbf{supervisor}}$ shown in Figure 13, we do not show the Modal Interface $\mathcal{C}$ 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 timing behavior and energy consumption. They would be developed with the same method and combined to the above contract $\mathcal{C}$ using conjunction.

D. Sub-contracting to suppliers

In this section, we apply the technique developed in Section VIII-E 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 duty of the designer is to specify an architecture “à la SysML”, as shown on Figure 14. Some comments are in order regarding this architecture. The considered instance of a parking garage consists of one entry gate, one exit gate, and one payment machine. Compare with the top-level specification of Table IX. The latter comprises a generic gate, a payment machine, and a supervisor, each one with its set of requirements. In contrast, the architecture of Figure 14 involves no supervisor. The supervisor is meant to be distributed among the two gates. The architecture of Figure 14 seems to be very loosely coupled. First, the PaymentMachine seems to be totally independent. In fact, the ticket that is inserted in the exit gate must coincide with the one issued by the PaymentMachine. It turns out that this reflects a missing assumption regarding the environment of the system (namely the user of the parking). Then, the two gates seem to have the

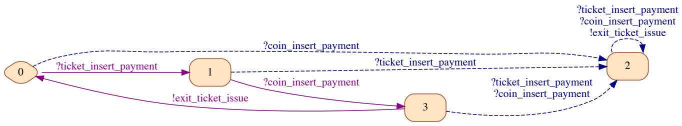

Figure 11. Chapter **payment** of the top-level requirements document translated into a Modal Interface $\mathcal{C}_{\mathbf{payment}}$.

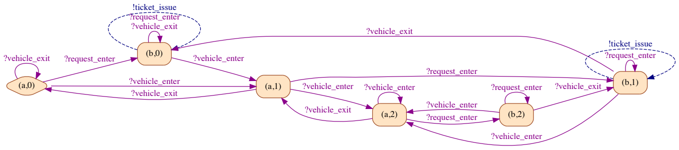

Figure 12. Modal Interface for $\mathrm{R}_{\mathrm{s}.3}$

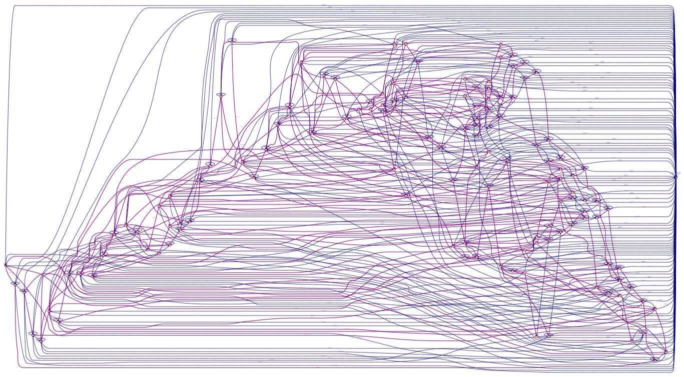

Figure 13. Chapter **supervisor** of the top-level requirements document translated into a Modal Interface $\mathcal{C}_{\mathbf{supervisor}}$, for a capacity of two for the parking garage.

shared input "?vehicle exit" as their only interaction. But this shared input is involved in requirement $\mathrm{R}_{\mathrm{s}.3}$, which forbids the entry if the parking is full.

In Figure 15 we show the result of applying, to the architecture of Figure 14, the Algorithm 1 developed in Section VIII-E, which yields by construction a refinement of the top-level contract $\mathcal{C}$ by a decomposition into local contracts:

$$\mathcal{C} \succeq \mathcal{C}(\Sigma_{\mathrm{EntryGate}}) \otimes \mathcal{C}(\Sigma_{\mathrm{ExitGate}}) \otimes \mathcal{C}(\Sigma_{\mathrm{PaymentMachine}}) \qquad (55)$$

Local contract $\mathcal{C}(\Sigma_{\mathrm{EntryGate}})$ is the more complex one because it involves counting—we have assumed a capacity of two to keep it simple. Remarkably enough, *the decomposition* (55) *involves small sub-systems compared to* $\mathcal{C}_{\mathbf{supervisor}}$ (Fig. 13) *and the global contract* $\mathcal{C}$*; the restriction operation is to be acknowledged for this strong reduction in size.*

E. The four "C"

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 [114]: Accuracy, Affordability, Boundedness, Class, Complexity, Completeness, Conciseness, Conformance, Consistency, Correctness, Criticality, Level, Orthogonality, Priority, Risk, Unambiguousness, 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.

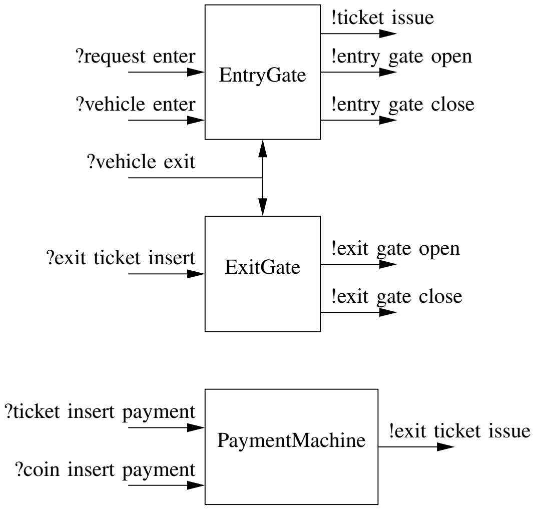

Figure 14. System architecture as specified by the designer.

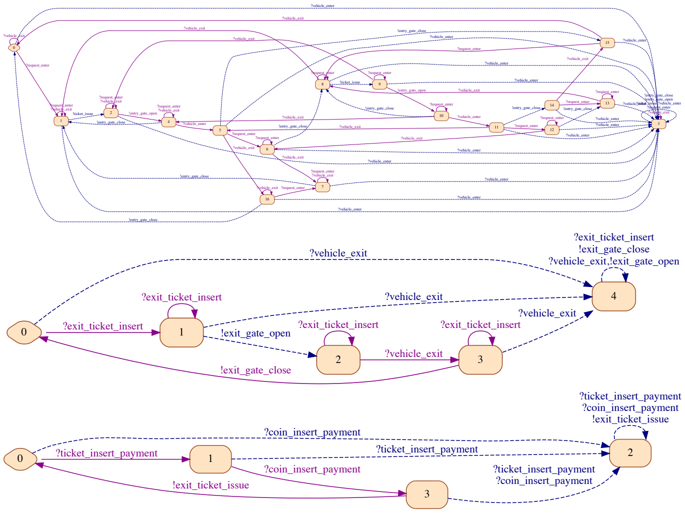

Figure 15. The three restrictions of the global contract $\mathcal{C}$ for the three subsystems EntryGate (top), ExitGate (mid), and PaymentMachine (bottom).

1) Consistency & Compatibility: Consistency and compatibility have been formally defined in the meta-theory, see Section V and Table IV 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.

It turns out that the existence of implementations is the formal definition of consistency according to the meta-theory. Clearly, the formal definition of consistency meets its common sense interpretation. We illustrate consistency on the top-level specification of Table IX. Referring to this table, we have strengthened $\mathrm{R}_{\mathrm{s}.6}$ and $\mathrm{R}_{\mathrm{s}.7}$ in some way. Lack of consistency is revealed by checking the mutual consistency of these two requirements with the requirement $\mathrm{R}_{\mathrm{g}.3}$ applied to the exit gate. After the following scenario:

```
exit_ticket_insert
exit_gate_open
exit_ticket_insert
```

event exit_gate_open has modality must in left-hand interface and modality cannot in right-hand interface.

According to the common meaning, an architecture of subsystems, 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—use cases can be operated as wished. As explained in Table IV of Section V, this is the formal definition of compatibility in the meta-theory. Again, the formal definition of compatibility meets its common sense interpretation.

2) Correctness: Correctness can only be defined with reference to another specification. We propose to translate "correctness" by one of the following properties, depending on the case (see Table IV for the notations):

- "is a correct implementation of", written $\models^{\mathrm{M}}$;
- "is a correct environment of", written $\models^{\mathrm{E}}$;
- "refines", written $\preceq$.

3) 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 Table IX, where $\mathrm{R}_{\mathrm{s}.6}$ is changed into "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:

```
exit_ticket_insert
exit_gate_open
vehicle_exit
exit_gate_close
exit_gate_open
```

which allows vehicles to 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 $\mathcal{C}$ is available. We propose to say that $\mathcal{C}'$ is *incomplete* with reference to $\mathcal{C}$, if

1) $\mathcal{C}'$ does not refine $\mathcal{C}$, but
2) there exists $\mathcal{C}'' \preceq \mathcal{C}'$ such that $\mathcal{C}''$ is consistent and compatible, and refines $\mathcal{C}$.

The rationale for this definition is that $\mathcal{C}'$ is not precise enough but can be made so by adding some more requirements. Note that $\mathcal{C}'$ is incomplete with reference to $\mathcal{C}$ if and only if $\mathcal{C}' \wedge \mathcal{C}$ is consistent and compatible. This criterion is of particular relevance when $\mathcal{C}' = \bigotimes_{i \in I} \mathcal{C}_i$ is an architecture of sub-contracts, where $\mathcal{C}_i$ is to be submitted to supplier i for independent development.

F. Discussion

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. Hence, systems engineers generally use several frameworks when expressing requirements. 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.[52]
- It can accommodate different concrete formalisms. In our example, we have blend 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 such as the four C's 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.[53]

XII. Contracts in the context of Autosar

In this section, we use Autosar as an industrial example of how to leverage contracts within an existing standard to extend it.

A. The Autosar context

Autosar[54] is a world-wide development partnership including almost all players in the automotive domain electronics

[52] 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.

[53] The framework of Assume/Guarantee contracts that is used in Section XII does not offer this, for two reasons. First, it lacks the quotient and, second, 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.

[54] http://www.autosar.org/

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 infrastructure based on standardized interfaces for the different layers. The AUTOSAR project has focused on the objectives of location independence, standardization of interfaces and portability of code. While these goals are undoubtedly of paramount importance, their achievement may not be sufficient for improving the quality of software systems. As for most other embedded system, car electronics is characterized by functional as well as non functional properties, assumptions and constraints [111]. Therefore, an AUTOSAR based development process benefits from adding contracts, as the this section will demonstrate.

B. The contract framework

Extensive use of Simulink/Stateflow modeling is performed for system design in the automobile sector. It is thus natural to try reusing this modeling style for defining contracts. Since A/G contracts of Section VII simply rely on specifying assertions regardless of how they are actually described, we use A/G contracts for this AUTOSAR case study. Since Simulink/Stateflow diagrams are very flexible, contracts can be expressed for all viewpoints (function, timing, safety). In turn, reasoning on such contracts is beyond the capability of automatic tools. Consequently, checking for implementation and refinement relations will be performed manually on small models. In contrast, the powerful contract algebra developed in the meta-theory is fully generic and does not depend on the nature and complexity of the considered models. The modularity it offers will thus be the main help offered by contract based design in this case study. As another feature of this this study, the distinction between "horizontal" and "vertical" contracts is used, see Section IV-D4. Finally, the following variation of A/G contracts is used.

Assertions consist of a timed extension of the "weakly synchronous" model (18), namely:

$$P \subseteq ((\Sigma \mapsto D) \cup \mathbb{R}_+)^* \cup ((\Sigma \mapsto D) \cup \mathbb{R}_+)^\omega \quad (56)$$

meaning that the assertion proceeds by a succession of steps consisting of either assignment of values to variables (including the special status "absent" for multiple-clocked assertions), or time progress. Composition is by intersection: $P_1 \times P_2 =_{\text{def}} P_1 \wedge P_2$, with alphabet equalization by inverse projection as in see Section VII-C. Following Section VII-B, *components* are tuples $M = (\Sigma^{\text{in}}, \Sigma^{\text{out}}, P)$, where $\Sigma = \Sigma^{\text{in}} \cup \Sigma^{\text{out}}$ is the decomposition of alphabet Σ into its *inputs* and *outputs*, and P is an assertion following (56). Components must be free of exception in the sense of Definition 4.

A *contract* is a tuple $\mathcal{C} = (\Sigma^{\text{in}}, \Sigma^{\text{out}}; A_s, A_w; G)$, where $\Sigma = \Sigma^{\text{in}} \cup \Sigma^{\text{out}}$ is the decomposition of alphabet Σ into its *inputs* and *outputs*. A_s and A_w, the *strong* and *weak assumptions,* and G, the *guarantees*, are assertions over Σ. The set $\mathcal{E}_\mathcal{C}$ of the legal environments for $\mathcal{C}$ collects all components E such that $E \subseteq A$. The set $\mathcal{M}_\mathcal{C}$ of all components implementing $\mathcal{C}$ is defined by $(A_s \wedge A_w) \times M \subseteq G$, meaning that implementations guarantee G under the condition that the environment satisfies the stronger condition $A_s \wedge A_w$. In other words, in this framework, legal environments are constrained by strong assumptions only, whereas the conjunction of strong and weak assumptions is required to guarantee G. Weak assumptions are useful for design iterations in case strict obligations for the context of use may not be realizable.

To conclude on this framework, its difference is only methodological since any contract with strong and weak assumptions, $\mathcal{C} = (\Sigma^{\text{in}}, \Sigma^{\text{out}}; A_s, A_w; G)$, is equivalent to an ordinary A/G contract $(\Sigma^{\text{in}}, \Sigma^{\text{out}}; A_s; G \vee \neg A_w)$. Thus, the reader is referred to Section VII. Adding time to the paradigm does not change anything to the relations or operators defined in that section.

However, adding time results in decidability and complexity issues. The latter problem can be fixed in part by using observers, see Sections V-G and VII. Other proofs can be carried on manually—this is feasible for undecidable properties when the needed reasoning only involves a small part of the analyzed system involving few locally interacting components. The added value of our contract framework is to support the combination of proofs into a validation for the overall system taking into account detailed design decisions captured in system and ECU configurations.

C. Exterior Light Management System

To illustrate the benefits of contract-based systems engineering in AUTOSAR, we consider as an example an excerpt of an Exterior Light Management for an automobile.[55] 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 requirements can be refined to contracts and discuss the added value of contracts for negotiations between OEM and suppliers. We will also illustrate the use of vertical contracts, e.g., to specify the expected latency of the communication solution and computations from the timing viewpoint, and a failure hypothesis in terms of reliability.

1) Function and timing: We begin our study by showing how AUTOSAR models can be enhanced with contracts.

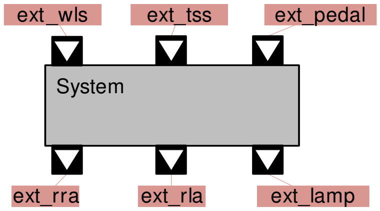

Figure 16. Virtual Functional Bus model

The Virtual Functional Bus model: Figure 16 shows the Virtual Functional Bus (VFB) model encompassing the exterior lights management. This software composition specifies the

[55] A case-study from the German SPES2020 project

interfaces exposed by the component to the sensors delivering driver inputs (brake pedal, warn light button, turn signal lever) and to the actuators controlling the lights. Ports with arrowheads denote an asynchronous data flow interface, which is a SenderReceiverInterface. Here it is a special kind called ServiceInterface, which means the data is provided by some local service located in the BasicSoftware stack of an ECU (Electronic Computing Unit). The requirements document of the system contains one timing-related requirement:

- R_1: If the driver presses the brake pedal, the brake lights must light not later than $25ms$.

We formalize in Table X this requirement as a contract by using a pattern-based contract specification language (terms in bold-face are keywords).

$\mathcal{C}_{R_1}$:

A_s	**whenever** ext_pedal **occurs** ext_pedal **does not occur during** [ext_pedal,ext_lamp]. ext_pedal **occurs sporadic with minperiod** 25ms.
A_w	**true**
G	**delay between** ext_pedal **and** ext_lamp **within** [0ms,25ms].

Table X
HORIZONTAL CONTRACT OF VFB MODEL

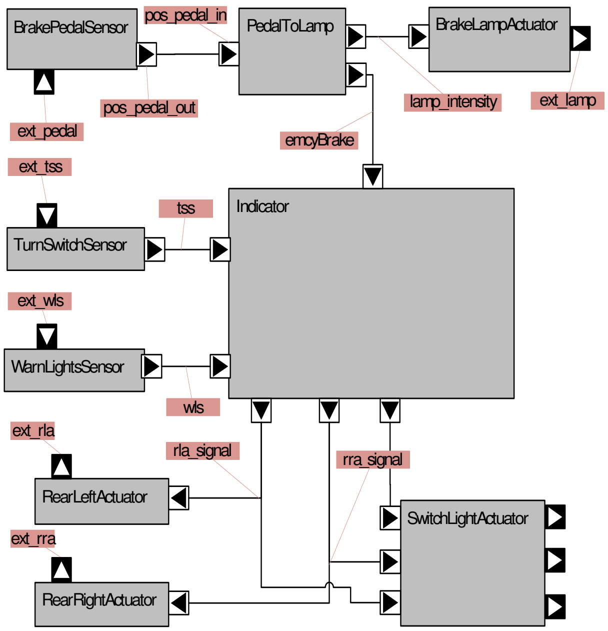

Figure 17. Decomposed Virtual Functional Bus model

Decomposed VFB Software composition: Figure 17 shows the decomposed VFB model. AssemblySwConnectors link the ports of parts of the composition and are subject to communication refinement. The components used in the composition and their functions are as follows:

- BrakePedalSensor receives the position of the brake pedal (ext_pedal) and distributes it to other components.
- PedalToLamp reads the brake pedal position (pos_pedal) and calculates requested brightness of the brake indicator lights (lamp_intensity).
- BrakeLampActuator reads the requested brightness of the brake indicator lights (lamp_intensity) and actuates the brake lamps accordingly (ext_lamp).
- TurnSwitchSensor reads the position of the lever and forwards it (tss).
- WarnLightsSensor reads status of the warn blink button and forwards it (wls).
- Indicator reads the status of the warn blink button, the lever position and cyclically toggles the left and right signals (rla_signal, rra_signal).
- Rear(Left|Right)Actuator reads its input signal ((rla|rra)_signal) and actuates the bulb (ext_(rla|rra)).
- SwitchLightActuator reads left and right signals ((rla|rra)_signal) and provides feedback on the dashboard.

Horizontal (functional) contracts associated to the VFB model: For the decomposed VFB model we derive horizontal contracts specifying the desired I/O behavior of the parts. As an example, for the component BrakePedalSensor, we state the contracts shown in Table XI. Contract $\mathcal{C}^f_{IO_{BPS}}$ specifies that, provided that the pedal is not "multiply pressed", then the pos_pedal message traverses the BrakePedalSensor, and then the wire following it, with some unspecified delay. In particular, no message is lost due to a miss by this block. Similar contracts are stated for all blocks of the VFB model, thus resulting in a set $\mathbf{C}^f_{IO}$ of contracts. For all AssemblySwConnectors of the composition, horizontal contracts are specified that are similar to that of $\mathcal{C}^f_{SIGNAL_{BPS}}$, resulting in a set $\mathbf{C}^f_{SIGNAL}$ of contracts. This contract scheme guarantees a reliable data transport by each wire of the VFB model, no matter how the communication is refined.

$\mathcal{C}^f_{IO_{BPS}}$:

A^f_s	**whenever** ext_pedal **occurs** ext_pedal **does not occur during** [ext_pedal,pos_pedal_out].
A^f_w	**true**
G^f	**whenever** ext_pedal **occurs** pos_pedal_out **occurs**.

$\mathcal{C}^f_{SIGNAL_{BPS}}$:

A^f_s	**whenever** pos_pedal_out **occurs** pos_pedal_out **does not occur during** [pos_pedal_out,pos_pedal_in].
A^f_w	**true**
G^f	**whenever** pos_pedal_out **occurs** pos_pedal_in **occurs**.

Table XI
A SAMPLE OF THE HORIZONTAL CONTRACTS OF THE DECOMPOSED VFB MODEL; SUPERSCRIPT f REFERS TO "FUNCTION".

So far we have stated contracts for each block and each wire of the VFB model. Compose the above contracts using contract composition (14), i.e., compute

$$\mathcal{C}^f_{VFB} = \left(\bigotimes\{\mathcal{C} \mid \mathcal{C} \in \mathbf{C}^f_{IO}\}\right) \otimes \left(\bigotimes\{\mathcal{C} \mid \mathcal{C} \in \mathbf{C}^f_{SIGNAL}\}\right) \tag{57}$$

The guarantee offered by overall contract $\mathcal{C}^f_{VFB}$ is the conjunction of the guarantees of the constitutive contracts. Hence, contract $\mathcal{C}^f_{VFB}$ guarantees that messages reliably traverse the VFB model with, however, an unspecified delay. On the other hand, contract composition (14) assumes the conjunction of all assumptions stated for each block—the guarantee offered by $\mathcal{C}^f_{VFB}$ and the strong assumption of $\mathcal{C}_{R_1}$ discharge all of these assumptions.

At this point, we observe that contract $\mathcal{C}^f_{VFB}$ involves no timing characteristics and, thus, functional contract $\mathcal{C}^f_{VFB}$ does not refine the initial contract $\mathcal{C}_{R_1}$. To overcome this, timing contracts are added in the next step, that will complement this functional contract with timing viewpoint based on expected performance characteristics of the computing platform.

Budgeting time using contracts: We now state additional contracts such as the ones shown in Table XII. Observe that, for those contracts weak assumptions are used, see Section XII-B regarding the motivations for doing this. Observe also that we repeated the functional strong assumption.[56] Again, stating similar contracts for each block and wire of the VFB model yields two sets $\mathbf{C}^t_{IO}$ and $\mathbf{C}^t_{SIGNAL}$ of contracts and we consider

$\mathcal{C}^t_{IO_{BPS}}$:	A^t_s	**whenever** ext_pedal **occurs** ext_pedal **does not occur during** [ext_pedal,pos_pedal_out].
	A^t_w	ext_pedal **occurs sporadic with minperiod** 15ms.
	G^t	**delay between** ext_pedal **and** pos_pedal_out **within** [0ms,15ms].

$\mathcal{C}^t_{SIGNAL_{BPS}}$:	A^t_s	**whenever** pos_pedal_out **occurs** pos_pedal_out **does not occur during** [pos_pedal_out,pos_pedal_in].
	A^t_w	pos_pedal_out **occurs sporadic with minperiod** 2ms.
	G^t	**delay between** pos_pedal_out **and** pos_pedal_in **within** [0ms,2ms].

Table XII
TIMING CONTRACT OF DECOMPOSED VFB MODEL; SUPERSCRIPT t INDICATES THAT THESE CONTRACTS DEAL WITH TIMING.

$$\begin{array}{rcl} \mathcal{C}^t_{VFB} & = & (\bigotimes\{\mathcal{C} \mid \mathcal{C} \in \mathbf{C}^t_{IO}\}) \\ & & \bigotimes(\bigotimes\{\mathcal{C} \mid \mathcal{C} \in \mathbf{C}^t_{SIGNAL}\}) \end{array} \tag{58}$$

The next step is to combine, for each component (block or wire) of the VFB model, the two functional and timing viewpoints using conjunction, i.e., to compute

$$\begin{array}{rcl} \mathcal{C}_{VFB} & = & (\bigotimes\left\{[\mathcal{C}^f \wedge \mathcal{C}^t] \mid (\mathcal{C}^f, \mathcal{C}^t) \in \mathbf{C}_{IO}\right\}) \\ & & \bigotimes(\bigotimes\left\{[\mathcal{C}^f \wedge \mathcal{C}^t] \mid (\mathcal{C}^f, \mathcal{C}^t) \in \mathbf{C}_{SIGNAL}\right\}) \end{array} \tag{59}$$

where $\mathbf{C}_{IO}$ collects, for each block, the pair of functional and timing contracts associated with it, and similarly for $\mathbf{C}_{SIGNAL}$. At this point, we observe that the architecture of the VFB model is a directed acyclic graph.

Consider the conjunction $\mathcal{C} = \mathcal{C}^f \wedge \mathcal{C}^t = (A, G)$ for each block or wire. As an example, focus on block IO_{BPS}, which is the first block of VFB, seen as a directed acyclic graph. Regarding the combination of strong and weak assumptions, we have $A = A^f_s \wedge A^t_w$, which is weaker than A_s, the assumption of $\mathcal{C}_{R_1}$ in Table X. Regarding guarantees, we have $G = G^f \wedge G^t$, which ensures that messages traverse the component without duplication or loss and with a delay less than the "minperiod" of the input signal (15ms). Also, guarantee G offered by this block is strong enough to discharge the assumption of the next wire when performing contract composition. Reproducing the same reasoning, inductively, along all blocks and wires of the VFB model seen as an acyclic graph, shows that

$$\mathcal{C}_{VFB} \quad \preceq \quad \mathcal{C}_{R_1} \tag{60}$$

Consequently, submitting the pairs of contracts (C^f, C^t) associated to each component for separate development, possibly by different suppliers, will ensure correct integration (provided implementations by the suppliers are correct with respect to their contracts). In particular, the component BrakePedalSensor belongs to the chassis domain and is implemented by a different supplier than the other components.

Discussion: The above step of our contract based design methodology leads to the following important notices:

1) So far our proof steps were all manual. Our contract based reasoning, however, was instrumental in ensuring that elementary proof steps were combined correctly. Reasoning based on intuition when component composition and viewpoint combination both occur is very likely to lead to wrong conclusions. *There is a strong added value in contract based reasoning even if elementary proof steps are manual.*

2) 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 V-G provides the formal support for this. For the pattern based language used here, a framework for checking refinement of contracts using an observer based strategy is described in [94].

3) The responsibility of establishing the sub-contracts submitted to the different suppliers is assigned to the OEM, in accordance with his role as integrator. Components belonging to different domains of a car (engine, chassis, etc.) are likely to originate from different vendors/suppliers. By using our above reasoning, the OEM can decompose its contract specifications, passing them to supplier(s). For example, the component BrakePedalSensor is the only one in the VFB model that belongs to the chassis domain and other components are developed by one or more different suppliers. Thus, in decomposition (59), the two sets of contracts $\mathbf{C}_{IO}$ and $\mathbf{C}_{SIGNAL}$ are partitioned,

[56] If we do not do this, then, when taking the conjunction of functional and timing contracts, the functional strong assumption gets absorbed by the strong assumption of the timing contract, since the latter is equal to T. See the discussion regarding Assume/Guarantee contracts with variable alphabets at the end of Section VII-C.

e.g., for subcontracting to two different suppliers:

$$\begin{aligned}
\mathcal{C}_{VFB}^{\text{supplier}_1} &= \left(\bigotimes\left\{\left[\mathcal{C}^f \wedge \mathcal{C}^t\right] \mid (\mathcal{C}^f, \mathcal{C}^t) \in \mathbf{C}^1_{IO}\right\}\right) \\
&\quad \otimes \left(\bigotimes\left\{\left[\mathcal{C}^f \wedge \mathcal{C}^t\right] \mid (\mathcal{C}^f, \mathcal{C}^t) \in \mathbf{C}^1_{SIGNAL}\right\}\right) \\
\mathcal{C}_{VFB}^{\text{supplier}_2} &= \left(\bigotimes\left\{\left[\mathcal{C}^f \wedge \mathcal{C}^t\right] \mid (\mathcal{C}^f, \mathcal{C}^t) \in \mathbf{C}^2_{IO}\right\}\right) \\
&\quad \otimes \left(\bigotimes\left\{\left[\mathcal{C}^f \wedge \mathcal{C}^t\right] \mid (\mathcal{C}^f, \mathcal{C}^t) \in \mathbf{C}^2_{SIGNAL}\right\}\right)
\end{aligned}$$

Each supplier can then safely develop its own sub-system, independently. We develop this in the next sub-section.

Implementation of software components and vertical assumptions: Each of the software components being part of the software system composition (cf. Figure 17) is a CompositionSoftwareComponents, as this allows the supplier to reuse already designed components. We thus consider one such component with its contract $\mathcal{C} = \mathcal{C}^f \wedge \mathcal{C}^t$ consisting of the functional contract $\mathcal{C}^f$, in conjunction with the timing contract $\mathcal{C}^t$. The supplier can then follow two different approaches for developing this component.

In a first approach, the supplier reuses a composition

$$\widehat{\mathcal{C}} \;=\; \bigotimes\nolimits_{i\in I} \mathcal{C}_i \tag{61}$$

of off-the-shelf components available from a library. Two situations can then occur:

- The easy case is when refinement holds: $\widehat{\mathcal{C}} \preceq \mathcal{C}$. This means that, for $\widehat{\mathcal{C}} = (\widehat{A}_s, \widehat{A}_w, \widehat{G})$, guarantees are strong enough and assumptions are weak enough to imply refinement with respect to the contract $\mathcal{C}$ specified by the OEM.
- A more realistic (but also more delicate) case is when the following weaker relations are satisfied:

$$\widehat{\mathcal{C}} \;\preceq\; \mathcal{C}^f \tag{62}$$

$$\widehat{G} \;\subseteq\; G \tag{63}$$

$$\widehat{A}_s \wedge \widehat{A}_w \;\supseteq\; A^f_s \wedge A^f_w \tag{64}$$

but the conditions requested on timing assumptions for refinement $\widehat{\mathcal{C}} \preceq \mathcal{C}$ to hold are not satisfied:

$$\widehat{A}_s \wedge \widehat{A}_w \;\not\supseteq\; A^t_s \wedge A^t_w \tag{65}$$

Under (65), composition from library $\widehat{\mathcal{C}}$ cannot be used as such. It can, however, still be used as a valid implementation provided that some additional missing vertical assumption regarding timing is stated, such that:

$$\widehat{A}_s \wedge \widehat{A}_w \;\supseteq\; A^t_s \wedge A^t_w \;\underbrace{\wedge\, \widehat{A}^t_w}_{\text{for negotiation}} \tag{66}$$

(66) requires a negotiation in which additional vertical assumption $\widehat{A}^t_w$ is returned back to the OEM as an additional performance requirement for the ECU platform. If this is accepted by the OEM, then development by the supplier using off-the-shelf components from the library (61) can proceed.

The alternative approach is to implement the component. Eventually, this results in a composition of AtomicSoftwareComponents. This component type has an internal behavior specification comprising so called Runnables, which are the smallest executable entities known to AUTOSAR. For each runnable, activation conditions are defined by means of RTEEvents controlled by the RuntimeEnvironment on the corresponding ECU, and data accesses referring ports of the owning component. An RTEEvent can be configured to occur either cyclically (time-triggered) or by the data arrival on some input port of the owning component (event-triggered).

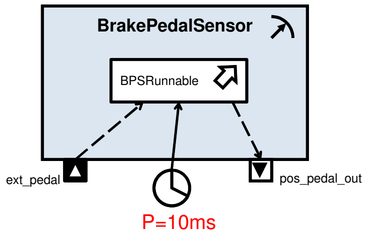

Figure 18. Internal behavior of BrakePedalSensor-component

$\widehat{\mathcal{C}}^f_{IB_{BPS}}$:	$\widehat{A}^f_s$	**whenever** ext_pedal **occurs** ext_pedal **does not occur during** [ext_pedal,pos_pedal_out].
	$\widehat{A}^f_w$	**true**
	$\widehat{G}^f$	**whenever** ext_pedal **occurs** pos_pedal_out **occurs**.

$\widehat{\mathcal{C}}^t_{IB_{BPS}}$:	$\widehat{A}^t_s$	**whenever** ext_pedal **occurs** ext_pedal **does not occur during** [ext_pedal,pos_pedal_out].
	$\widehat{A}^t_w$	ext_pedal **occurs sporadic with minperiod** 10ms. BPSRunnable#act **occurs each** 10ms. **delay between** BPSRunnable#act **and** pos_pedal_out **within** [0ms,10ms].
	$\widehat{G}^t$	**delay between** ext_pedal **and** pos_pedal_out **within** [0ms,10ms + 10ms].

$\widehat{\mathcal{C}}'^t_{IB_{BPS}}$:	$\widehat{A}'^t_s$	**whenever** ext_pedal **occurs** ext_pedal **does not occur during** [ext_pedal,pos_pedal_out].
	$\widehat{A}'^t_w$	ext_pedal **occurs sporadic with minperiod** 10ms. BPSRunnable#act **occurs each** 10ms. **delay between** BPSRunnable#act **and** pos_pedal_out **within** [0ms,5ms].
	$\widehat{G}'^t$	**delay between** ext_pedal **and** pos_pedal_out **within** [0ms,10ms + 5ms].

Table XIII
CONTRACTS FOR INTERNAL BEHAVIOR OF BRAKEPEDALSENSOR-COMPONENT

Figure 18 depicts the internal behavior of component BrakePedalSensor. This component consists of a single runnable BPSRunnable, which is activated cyclically with a period of $10ms$. Upon its activation it reads the sensor values from the ECU abstraction component and writes the position of the brake pedal on its provided Sender/Receiver port. The timing behavior of the component owning a time-triggered runnable can be characterized by the contract shown in Table XIII, top.

We now focus on the second situation in which the supplier wants to reuse components from a library to implement contract $\mathcal{C} = \mathcal{C}^f_{IO_{BPS}} \wedge \mathcal{C}^t_{IO_{BPS}}$. We consider the case where a component BrakePedalSensor has already been implemented and characterized by $\widehat{\mathcal{C}} = \widehat{\mathcal{C}}^f_{IB_{BPS}} \wedge \widehat{\mathcal{C}}^t_{IB_{BPS}}$ Now, observe that relation (62) holds. However, $\widehat{G} \not\subseteq G$. Taking a closer look at $\widehat{A}^t_w$, we see it assumes a worst case response time for BPSRunnable. Thus, the guarantee $\widehat{G}^t$ can be strengthened if weak assumption $\widehat{A}^t_w$ is replaced by $\widehat{A}'^t_w$ ensuring a shorter response time. The supplier would then consider the contract $\widehat{\mathcal{C}}' = \widehat{\mathcal{C}}^f_{IB_{BPS}} \wedge \widehat{\mathcal{C}}'^t_{IB_{BPS}}$, where the re-adjusted contract $\widehat{\mathcal{C}}'^t_{IB_{BPS}}$ corresponds to assuming $\widehat{A}'^t_w$ and guaranteeing $\widehat{G}'^t$. For that modified contract, relations (62–65) hold. However, strengthening an assumption is not a refinement. Therefore, applying this modification cannot be performed by the supplier on its own but rather requires negotiation with the OEM. If the OEM is willing to accept the stronger assumption $\widehat{A}'^t_w$, the component from the libary can be used and the modified contract of the supplier is refined: $\widehat{\mathcal{C}}' \preceq \mathcal{C}'^{\text{supplier}_1}_{VFB}$.

The next process steps in the AUTOSAR methodology, called *System configuration* and *ECU configuration* define the target architecture (ECUs and communication network) as well as the deployment of the application software components on that distributed architecture. Thus, satisfaction of the contracts specified for the application needs to be verified for the given architecture and deployment (cf. IV-D4). We develop this in the next two sub-sections.

ECU configuration and vertical contracts: During the following discussion we assume that for each component implementation a contract similar to $\widehat{\mathcal{C}}'$ has been specified and possibly an associated $\widehat{A}'^t_w$ has been negotiated. Observe, that $\widehat{A}'^t_w$ may invalidate refinement of system requirements by the VFB contract. Consider the example above. $\widehat{A}'^t_w$ assumes an activation of BPSRunnable each 10ms and a worst case response time less than 5ms. This additional weak assumption is now part of $\mathcal{C}'^{\text{supplier}_1}_{VFB}$. However, it is neither discharged by any guarantee of $\mathcal{C}'_{VFB}$, nor by the strong assumption of $\mathcal{C}_{R_1}$. Thus, we need to verify that 1/ all vertical assumptions collected during previous steps are discharged during system- and ECU configuration and 2/ all guarantees are met by software components and connectors. For both tasks state-of-the-art verification techniques can be used.

As an example, focus on the vertical assumption $\widehat{A}'^t_w$ of the BrakePedalSensor-component and consider the ECU configuration depicted in Figure 19. The ECU hosts the two components BrakePedalSensor and PedalToLamp of the VFB model[57]. These components are integrated with a layered basic software stack on an ECU. That stack consists of multiple BasicSoftwareModules (BSWM). Each of these modules has a set of standardized interfaces and is implemented by a set

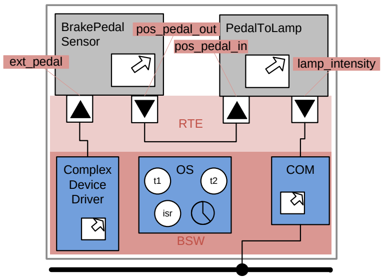

Figure 19. Extracting platform contracts from ECU configuration

of BswModuleEntities. These are analogous to the Runnables of application software components and are either executed by a dedicated OS task or interrupt service routine. Therefore, we expect BasicSoftwareModules to be characterized by contracts in a way similar to the internal behavior of application components (cf. contracts in Table XIII). In case a BSWM has a BswModuleEntity that is not mapped to some task or interrupt service routine, that entity is an “ordinary” function called within the context of another BswModuleEntity or Runnable. In the example ECU configuration two BSWMs are shown:

- A ComplexDeviceDriver, which has direct access to the sensor of the ECU that detects position of the brake pedal.
- An AUTOSAR COM module and a CAN interface (not shown here).

The components and BSWMs own a single runnable each, bsw entity respectively. The ECU configuration maps BPSRunnable to task t1 of the operating system (OS) and PTLRunnable to task t2. The BswModuleEntity of the ComplexDeviceDriver is mapped to an interrupt service routine, while the entity of the COM module is simply a function for sending a signal. That function is used by the AUTOSAR runtime environment (RTE) to realize the communication specified on VFB-level by means of the connectors. For each task of the OS the ECU configuration specifies at least its priority and wether it is preemptable. The activation of a task is specified by linking an alarm of the OS to that task. Those alarms can also be configured to occur cyclically. For each cyclic alarm we generate a contract $\mathcal{P}_{ALARM_i}$, with trivial strong and weak assumptions and as guarantee the cycle time of the alarm. The example ECU configuration contains a single alarm, whose contract is listed in Table XIV, top. Furthermore,

$\mathcal{P}_{ALARM_1}$:

A_s	**true**.
A_w	**true**.
G	$Alarm_i$ **occurs each** 10ms.

$\mathcal{P}_{BPSRunnable}$:

A_s	**true**.
A_w	**true**.
G	**delay between** BPSRunnable#act **and** pos_pedal_out **within** [1ms,4ms].

Table XIV
PLATFORM CONTRACTS OBAINED BY ANALYSIS OF ECU CONFIGURATION

[57] To simplify, we omit port emcyBrake

we can extract from the AUTOSAR model of the component implementation a set of execution time specifications for each Runnable owned by the component. Again, the same applies to BswModuleEntities. With this information we can create a task network and obtain response times for each Runnable and BswModuleEntity (if executed by some task or interrupt service routine) using tools like SymTA/S[58] or chronVAL[59]. Afterwards, we generate for each Runnable and BswModuleEntity (if executed by task or ISR) a contract $\mathcal{P}_{Runnable_i}$, with stong and weak assumption T and as guarantee the reponse time interval determined by analysis. For the ECU configuration sketched in Figure 19, BPSRunnable has a response time that lies in the interval $[1ms, 4ms]$, resulting in the contract shown in Table XIV, bottom. Similar contracts are stated for all tasks and alarms extracted from the ECU configurations. We then compose these contracts and the contracts of the BSWMs, which yields the contract $\mathcal{P}$ characterizing the execution platform. Following the approach outlined in Section IV-D4 we compose the VFB contract with the platform contract and check for refinement, i.e.

$$\mathcal{C}'_{VFB} \otimes \mathcal{P} \quad \preceq \quad \mathcal{C}'_{VFB} \tag{67}$$

The vertical assumption $\widehat{A}'^{t}_{w}$ is discharged in the composition of the VFB and platform contracts and refinement holds. Furthermore, with $\widehat{A}'^{t}_{w}$ being discharged refinement of the system requirement holds again:

$$\mathcal{C}'_{VFB} \otimes \mathcal{P} \quad \preceq \quad \mathcal{C}_{R_1} \tag{68}$$

The approach outlined here to analyze the mapping of application software components and their associated contracts on an ECU network by creating contracts from ECU configurations and results of response time analysis, has been implemented in a framework. That framework includes a meta-model supporting contract-based design and enables to 1/ specify contracts for components in foreign models and 2/ check for refinement between contracts belonging to different models [24].

Inter-ECU communication: During system configuration the leaves of the hierarchical software composition called AtomicSoftwareComponents are mapped to ECUs of the target architecture. Thus, we know for each connector of the VFB model, whether it must be implemented by the communication network interconnecting the ECUs.

For both, intra- and inter-ECU communication the design must be verified against the corresponding contracts of the VFB model, i.e. the set of contracts

$$\left\{ \left[\mathcal{C}^f \wedge \mathcal{C}^t\right] \,\middle|\, (\mathcal{C}^f, \mathcal{C}^t) \in \mathbf{C}_{SIGNAL} \right\} \tag{69}$$

Again, available analysis techniques can be used. Suppose, a signal from the VFB model is mapped to a frame transmitted on a CAN bus. The OEM can then for each such contract use the combination of strong and weak assumptions $A = A^f_s \wedge A^t_w$ as specification of the minimal inter-arrival times of subsequent occurrences of the signal. Taking into account the priority of each CAN frame given by its identifier, the response time of each signal can be computed and compared against the guarantee $G = G^f \wedge G^t$.

If the communication network, whose configuration is part of the AUTOSAR system configuration, is successfully verified against the contracts defined in (69), deployment of the VFB model on the target architecture is correct with regard to the VFB contract and the design fulfills the initial system requirement $\mathcal{C}_{R_1}$.

Discussion: The above development highlights the methodological value of contracts for system design in AUTOSAR context at later design stages. First, both contract conjunction and composition are used. Then, modifications that are not refinements are pinpointed, which forces the supplier to engage on a negotiation with the OEM regarding system specifications. Without the solid support of contracts and its rules, such kind of reasoning would be error prone.

2) Safety: The virtual function bus (VFB) concept was intended to enable a distributed development of software components independently from the underlying hardware architecture. The specification of the interfaces in the 3.x releases did not support the kind of non functional property necessary to develop safety aspects of critical systems. This fact lead to the development of the previously mentioned *Timing Extension* and *Safety Extension* to Methodology and Templates. The safety extension directly addresses the use of contracts for the specification of safety relevant properties driven by the need for compliance to the upcoming ISO26262 standard. The ISO26262 not only requires comprehensive checking of requirements but also states explicitly the need for using assumptions on the environment to develop a safety relevant element on the supplier side ("safety element out of context"). In particular the standard distinguishes between:

- A *functional safety concept* comprising the functional safety requirements, their allocation to architectural elements, and their interaction necessary to achieve the safety goals. This is used to demonstrate that the risks related to an item of a vehicle are addressed by an adequate functional architecture.
- A *technical safety concept* used to demonstrate that the functional safety concept is addressed by an adequate technical solution.

As a simple example for this approach we take the functional architecture of Figure 20 showing the decomposition of the SafeBrakeIndication function into the "normal" BrakeIndication function and the SignalingFailureDetection safety function in case the BrakeIndication fails. The corresponding safety goal requires that a loss of the braking indication function shall be detected. This is formalized as contract $\mathcal{C}_{S_0}$, shown in Table XV. This contract states that under a given failure hypothesis ("no double failure") either the brake lamp is activated or the driver is warned by the SignalingFailureDetection function. This contract is attached to the function SafeBrakeIndication.

For the subfunction BrakeIndication we show in Table XVI a contract $\mathcal{C}_{S_1}$ stating the correct behavior in the absence

[58] http://www.symtavision.com/symtas.html
[59] http://www.inchron.com/chronval.html

$\mathcal{C}_{S_0}$:

A_s	**true**
A_w	**Only one of** Fail(BrakeIndication) **and** Fail(SignalFailureDetection) **occurs**
G	**Whenever** BreakRequest **occurs then** BreakIndication **or** SignalFailureDetection **is performed**.

Table XV
HORIZONTAL CONTRACT REPRESENTING FUNCTIONAL SAFETY REQUIREMENT

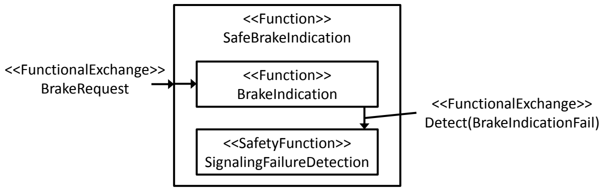

Figure 20. Functional architecture including safety function

of a failure of this function: For the SignalingFailureDetection the associated contract states that a detected failure of the sub-function BrakeIndication is correctly indicated under the assumption that the provided function itself is not impacted by a failure, see Table XVII. An easy manual inspection shows that the composition of $\mathcal{C}_{S_1} \otimes \mathcal{C}_{S_2}$ refines $\mathcal{C}_{S_0}$, i.e., we have formally validated the functional safety concept.

$\mathcal{C}_{S_1}$:

A_s	**true**
A_w	**Absence of** Fail(BrakeIndication)
G	**Whenever** BrakeRequest **occurs then** BrakeIndication **is performed**.

Table XVI
HORIZONTAL CONTRACT FOR FUNCTION OF BRAKING LAMP

$\mathcal{C}_{S_2}$:

A_s	**true**
A_w	**Absence of** fail(SignalingFailureDetection)
G	**Whenever** detect(fail(BrakeIndication)) SignalingFailureDetection **is performed**.

Table XVII
HORIZONTAL CONTRACT FOR SAFETY FUNCTION FOR BRAKING LAMP

The next step consists in allocating these functions to software components on the VFB and developing a simple technical safety concept. The corresponding technical architecture is shown in Figure 21. For the sake of simplicity we only consider corrupted data while transmitting under failure mode. The affected communication links are marked in blue in Figure 21. Thus, fail(BrakeIndication) is mapped to fail(pos_pedal) and fail(lamp_intensity). In the same way fail(SignalingFailureDetection) is mapped to fail(mal_indication). This mapping is not further elaborated in this paper.

For defining the technical safety concept, so-called *propagation contracts* are used. Propagation contracts express how failures (from the environment of the component) are processed by the component. According to the ISO 26262 (see Figure 22) Faults, Errors and Failures are distinguished by the patterns used for the contracts.

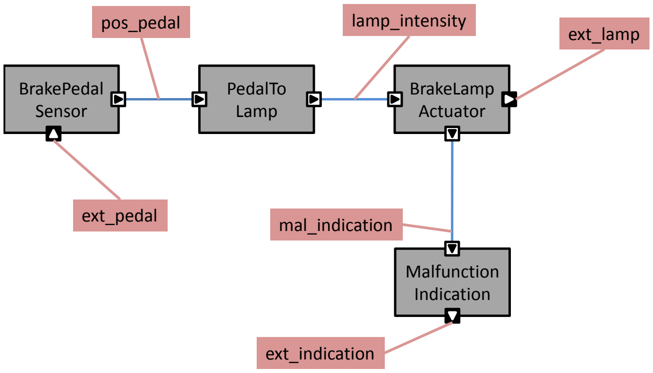

Figure 21. Extract of Indicator System

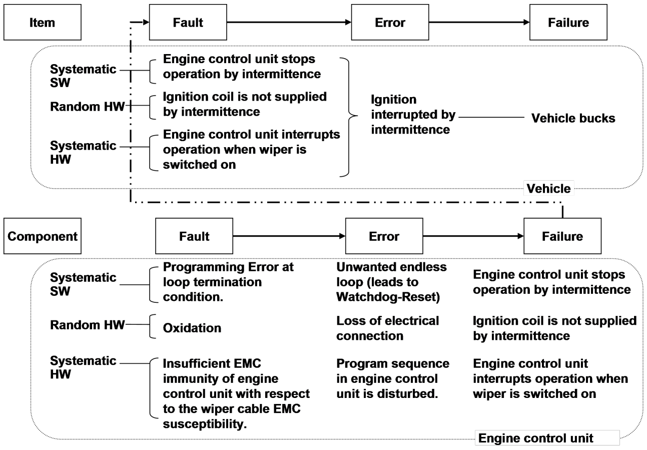
Item
Fault
Error
Failure
Systematic SW
Engine control unit stops operation by intermittence
Random HW
Ignition coil is not supplied by intermittence
Ignition interrupted by intermittence
Vehicle bucks
Systematic HW
Engine control unit interrupts operation when wiper is switched on
Vehicle
Component
Fault
Error
Failure
Systematic SW
Programming Error at loop termination condition.
Unwanted endless loop (leads to Watchdog-Reset)
Engine control unit stops operation by intermittence
Random HW
Oxidation
Loss of electrical connection
Ignition coil is not supplied by intermittence
Systematic HW
Insufficient EMC immunity of engine control unit with respect to the wiper cable EMC susceptibility.
Program sequence in engine control unit is disturbed.
Engine control unit interrupts operation when wiper is switched on
Engine control unit

Figure 22. The role of Faults, Errors, and failures

For instance, the contract $\mathcal{C}_{S_4}$ of Table XVIII states that the error state of the PedalToLamp component is caused by a failure in the communication.

$\mathcal{C}_{S_4}$:

A_s	**true**
A_w	**true**
G	**Mode** PedalToLampError **is caused by** fail(pos_pedal).

Table XVIII
HORIZONTAL CONTRACT FOR SAFETY FUNCTION FOR BRAKING LAMP

The nominal behavior is again given by the contract with an assumption about the absence of the communication failure, see Table XIX.

Similarly, the error state of the BreakLampActuator is caused by a failure on the second communication link (fail(lamp_intensity)) or by a failure propagated from the PedalToLamp component, see Table XX. The nominal behavior of the

$\mathcal{C}_{S_5}$:

A_s	**true**
A_w	**Absence of** fail(pos_pedal)
G	**Whenever** pos_pedal **occurs then** lamp_intensity **occurs**.

Table XIX
EXEMPLARY CONTRACT FOR NOMINAL BEHAVIOR SPECIFICATION

BrakePedalSensor and the MalfunctionIndicator is formed similarly to $\mathcal{C}_{S_5}$.

$\mathcal{C}_{S_6}$:

A_s	**true**
A_w	**true**
G	**Mode** BreakLampActuatorError **is caused by** fail(lamp_intensity) **or mode** PedalToLampError.

Table XX
HORIZONTAL CONTRACT FOR SAFETY FUNCTION FOR BRAKING LAMP

The composition of contracts $\mathcal{C}_{S_4} \dots \mathcal{C}_{S_7}$ together with the not stated simple nominal contracts entails the contract $\mathcal{C}_{S_8}$ shown on Table XXI stating the functional safety requirement.

$\mathcal{C}_{S_8}$:

A_s	**true**
A_w	**only one of** faila failb failc
G	**Whenever** ext_pedal **occurs then** ext_lamp **or** ext_indication **occurs**.

Table XXI
CONTRACT STATING THE TECHNICAL SAFETY REQUIREMENT

Contract $\mathcal{C}_{S_8}$ is directly related to the functional safety goal and includes the allocation to a technical safety concept. This way a consistency between the functional and the technical safety concept has been established. Furthermore, thanks to the formal pattern-based specification language, the refinement relations ensure that the concepts themselves are correct.

To conclude, let us mention other safety related use-cases which benefit from contract based specification such as the automatic generation of fault-trees or FMEA tables as well as the safe-state management.

D. Integrating Contracts in AUTOSAR

An important challenge is to smoothly integrate the use of contracts as part of AUTOSAR methodology. We advocate a migration in two steps.

Today—Offering a Smooth Integration of Contract-Based System Engineering with AUTOSAR*:* The metamodel underlying the CESAR reference technology platform has been carefully designed to support a seamless design flow from the derivation of high-level requirements of new automotive functions to the level of detailed application design defined as the entry point in the AUTOSAR design methodology. The key for this smooth integration has been laid by AUTOSAR itself, by enforcing a clean separation between application development on the one side, and deployment to distributed target architectures on the other side. Regarding layered design, the joint scope of CESAR and AUTOSAR ranges from systems-of-systems levels for Car2X applications, to the introduction of new automotive functions, and down to the detailed design level. As illustrated in the previous section, such phases would build on established design processes and currently used design tools, where contracts and viewpoints are conservatively added as additional flavor to existing design methodologies. Full hand-over to AUTOSAR takes place at the detailed design stage, where component contracts get implemented using the AUTOSAR component model, thus providing the entry point for the AUTOSAR design methodology.

Following the AUTOSAR flow, system configurations and ECU configurations describe the detailed target hardware. We can then automatically extract from system and ECU configurations all information for performing real-time analysis. Vertical assumptions on the component level can be derived by back-annotation from the underlying target hardware. Alternatively, vertical assumptions can be seen as constraining the target architecture (regarding its timing characteristics) and can thus be used to constrain the design space for possible target architectures. Finally, when deploying components to ECUs, communications of the virtual functional bus are mapped to the inter-component communications on the physical architecture.

The Future—Integrating Contracts in AUTOSAR*:* 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. However, the current release was not designed for extensibility: there is no established notion of viewpoints and, hence, no anchor to easily migrate from the current setting to one supporting multiple viewpoints. As pointed out before, the decision as to which viewpoints should or should not be anchored is a business decision. However, given that ISO 26262 has become binding, we see the need for a systematic integration of the safety viewpoint.

In fact, a safety extension has already been proposed for inclusion in AUTOSAR. It allows to use contracts for the encapsulation of Basic SoftWare (BSW) modules, application software components, entities of the hardware architecture (like ECUs and buses) and the characterization of communication services. Such encapsulation would, for instance, distinguish between BSW modules that (1) contribute to safety ("safety mechanisms") and (2) those that could potentially impact system's safety (by providing their failure propagation behavior). A typical example for (1) would be the function inhibition manager that maintains function inhibitions whenever, for instance, implausible outputs are generated. A typical example for (2) would be an error that occurs in the communication services or an error that is generated in communication H/W that impacts communication services.

The objective of the extension is to provide an ISO 26262 compliant traceability. This allows, for instance, to deal with compliance between functional safety (where functions, hazards and safety functions are collected in the functional model) and technical safety of the AUTOSAR architecture instance (guaranteeing that the AUTOSAR implementation—using given safety functions of the AUTOSAR architecture—indeed implements the identified functional safety concept).

Since timing and safety viewpoints will be supported in AUTOSAR it is at least worthwhile to consider introducing the concepts of viewpoints systematically. Doing so will ease their activation at times when other market factors such as power-management mandate their introduction.

E. Summary and discussion

We have proposed a smooth path for integrating contracts in the context of AUTOSAR at the level of detailed design. We believe that the main aspects for consideration while introducing contracts are the handling of time budgets and the specification of failure modes and failure propagation, in combination with the functional aspect itself. Vertical contracts were instrumental in handling time budgets. Such contracts relate two different layers of the AUTOSAR framework.

XIII. CONCLUSION

This paper presented past and recent results as well as novel advances in the area of contracts and their theory. By encompassing (functional and non-functional) behaviors, the notion of contract we considered here represents a significant step beyond the one originally developed in the software engineering community.

A. What contracts can do for the designer

This paper demonstrates that contracts offer a number of advantages:

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.

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 of Section XII demonstrated, using contracts makes semi-formal design safer. Small analysis steps are within the reach of human reasoning. A valid, system wide, analysis for both component-based and refinement-based designs is typically beyond the reach of human reasoning. Contract based design enables it.

Contracts improve requirement engineering: As illustrated in the Parking Garage example of Section XI, 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 can be used in any design process: Contracts offer a "orthogonal" support for all methodologies and can be used in any flow as a supporting technology in composing and refining designs.

B. Status of research

The area of contracts 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 are seen as two players in a game. Modalities aimed to offer more expressive logics were born at the boundary between logics and formal verification. Contracts as a philosophy originated both from software engineering and formal verification communities, with the paradigms of Pre-condition/Post-condition or Assume/Guarantee. It is 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 paper was intended to show the power of a unified theoretical background for contracts, the use of contracts in present methodologies and the challenges for its effective applications in future applications. The mathematical elegance of the concepts underpinning this area provides confidence in a sustained continuation of the research effort.

C. Status of practice

The use of contract-based techniques in system design is in its infancy in industry. Further maturation is needed for its paradigm and concepts to become, on one side, clear and simple to be widely accepted by engineers in their day-to-day work and, on the other side, developed enough to allow for the development of tools for automatic verification and synthesis. While powerful contract-based proof-of-concept tools are being experimented—some of them were presented in this paper—the robustness of the tools and the underlying techniques is still weak, and contract-based design flows and methodologies are not yet fully developed nor mature.

D. The way forward

The ability of contracts to accommodate semi-formal and formal methodologies should enable a smooth and rapid migration from theory and proof-of-concepts to robust flows and methodologies. The need for jointly developing new systems while considering issues of intellectual property will make it attractive to rely on contracts in supplier chains. In our opinion, contracts are primarily helpful for early stages of system design and particularly requirement engineering, where formal methods are desperately needed to support distributed and concurrent development by independent actors.

We have illustrated in this paper 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 presented in Section XI) can be used in combination with both manual reasoning and dedicated formal verification engines (e.g., for targeting the timing viewpoint or the safety viewpoint) as the basis for future development that will make contracts main stream.

REFERENCES

[1] Martín Abadi and Leslie Lamport. Conjoining specifications. *ACM Trans. Program. Lang. Syst.*, 17(3):507–534, 1995.

[2] Yael Abarbanel, Ilan Beer, Leonid Gluhovsky, Sharon Keidar, and Yaron Wolfsthal. FoCs - Automatic Generation of Simulation Checkers from Formal Specifications. In E. Emerson and A. Sistla, editors, *Computer Aided Verification*, volume 1855 of *Lecture Notes in Computer Science*, pages 538–542. Springer Berlin / Heidelberg, 2000.

[3] B. Thomas Adler, Luca de Alfaro, Leandro Dias da Silva, Marco Faella, Axel Legay, Vishwanath Raman, and Pritam Roy. Ticc: A Tool for Interface Compatibility and Composition. In *Proc. of the 18th International Conference on Computer Aided Verification (CAV'06)*, volume 4144 of *Lecture Notes in Computer Science*, pages 59–62. Springer, 2006.

[4] Albert Benveniste and Benoît Caillaud and Jean-Baptiste Raclet. Application of Interface Theories to the Separate Compilation of Synchronous Programs. In *IEEE Conf. on Decision and Control*, dec 2012.

[5] Luca De Alfaro and Thomas A. Henzinger. Interface-based design. In *In Engineering Theories of Software Intensive Systems, proceedings of the Marktoberdorf Summer School*. Kluwer, 2004.

[6] Rajeev Alur and David L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235, 1994.

[7] Rajeev Alur, Limor Fix, and Thomas A. Henzinger. A determinizable class of timed automata. In David L. Dill, editor, *CAV*, volume 818 of *Lecture Notes in Computer Science*, pages 1–13. Springer, 1994.

[8] Rajeev Alur, Limor Fix, and Thomas A. Henzinger. Event-clock automata: A determinizable class of timed automata. *Theor. Comput. Sci.*, 211(1-2):253–273, 1999.

[9] Rajeev Alur, Thomas A. Henzinger, and Orna Kupferman. Alternating-time temporal logic. *J. ACM*, 49(5):672–713, 2002.

[10] Rajeev Alur, Thomas A. Henzinger, Orna Kupferman, and Moshe Y. Vardi. Alternating refinement relations. In *Proc. of the 9th International Conference on Concurrency Theory (CONCUR'98)*, volume 1466 of *Lecture Notes in Computer Science*, pages 163–178. Springer, 1998.

[11] Adam Antonik, Michael Huth, Kim G. Larsen, Ulrik Nyman, and Andrzej Wasowski. 20 Years of Modal and Mixed Specifications. *Bulletin of European Association of Theoretical Computer Science*, 1(94), 2008.

[12] Adam Antonik, Michael Huth, Kim Guldstrand Larsen, Ulrik Nyman, and Andrzej Wasowski. Complexity of Decision Problems for Mixed and Modal Specifications. In *FoSSaCS*, pages 112–126, 2008.

[13] Road vehicles – functional safety. Standard ISO 26262.

[14] Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking*. MIT Press, Cambridge, 2008.

[15] Felice Balarin, Jerry R. Burch, Luciano Lavagno, Yosinori Watanabe, Roberto Passerone, and Alberto L. Sangiovanni-Vincentelli. Constraints specification at higher levels of abstraction. In *Proceedings of the Sixth IEEE International High-Level Design Validation and Test Workshop (HLDVT01)*, pages 129–133, Monterey, CA, November 7–9, 2001. IEEE Computer Society, Los Alamitos, CA, USA.

[16] Felice Balarin, Abhijit Davare, Massimiliano D'Angelo, Douglas Densmore, Trevor Meyerowitz, Roberto Passerone, Alessandro Pinto, Alberto Sangiovanni-Vincentelli, Alena Simalatsar, Yoshinori Watanabe, Guang Yang, and Qi Zhu. Platform-based design and frameworks: METROPOLIS and METRO II. In Gabriela Nicolescu and Pieter J. Mosterman, editors, *Model-Based Design for Embedded Systems*, chapter 10, page 259. CRC Press, Taylor and Francis Group, Boca Raton, London, New York, November 2009.

[17] Felice Balarin and Roberto Passerone. Functional verification methodology based on formal interface specification and transactor generation. In *Proceedings of the Conference on Design, Automation and Test in Europe (DATE06)*, pages 1013–1018, Munich, Germany, March 6–10, 2006. European Design and Automation Association, 3001 Leuven, Belgium.

[18] Felice Balarin and Roberto Passerone. Specification, synthesis and simulation of transactor processes. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 26(10):1749–1762, October 2007.

[19] Felice Balarin, Roberto Passerone, Alessandro Pinto, and Alberto L. Sangiovanni-Vincentelli. A formal approach to system level design: Metamodels and unified design environments. In *Proceedings of the Third ACM and IEEE International Conference on Formal Methods and Models for Co-Design (MEMOCODE05)*, pages 155–163, Verona, Italy, July 11–14, 2005. IEEE Computer Society, Los Alamitos, CA, USA.

[20] Felice Balarin, Yosinori Watanabe, Harry Hsieh, Luciano Lavagno, Claudio Passerone, and Alberto L. Sangiovanni-Vincentelli. Metropolis: An Integrated Electronic System Design Environment. *IEEE Computer*, 36(4):45–52, 2003.

[21] Krishnakumar Balasubramanian, Aniruddha Gokhale, Gabor Karsai, Janos Sztipanovits, and Sandeep Neema. Developing applications using model-driven design environments. *IEEE Computer*, 39(2):33–40, 2006.

[22] Sebastian S. Bauer, Alexandre David, Rolf Hennicker, Kim Guldstrand Larsen, Axel Legay, Ulrik Nyman, and Andrzej Wasowski. Moving from specifications to contracts in component-based design. In Juan de Lara and Andrea Zisman, editors, *FASE*, volume 7212 of *Lecture Notes in Computer Science*, pages 43–58. Springer, 2012.

[23] Sebastian S. Bauer, Philip Mayer, Andreas Schroeder, and Rolf Hennicker. On Weak Modal Compatibility, Refinement, and the MIO Workbench. In *Proc. of 16th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'10)*, volume 6015 of *Lecture Notes in Computer Science*, pages 175–189. Springer, 2010.

[24] Andreas Baumgart, Eckard Böde, Matthias Büker, Werner Damm, Günter Ehmen, Tayfun Gezgin, Stefan Henkler, Hardi Hungar, Bernhard Josko, Markus Oertel, Thomas Peikenkamp, Philipp Reinkemeier, Ingo Stierand, and Raphael Weber. Architecture Modeling. Technical report, OFFIS, March 2011.

[25] Nikola Benes, Jan Kretínský, Kim Guldstrand Larsen, and Jirí Srba. Checking Thorough Refinement on Modal Transition Systems Is EXPTIME-Complete. In Martin Leucker and Carroll Morgan, editors, *ICTAC*, volume 5684 of *Lecture Notes in Computer Science*, pages 112–126. Springer, 2009.

[26] Nikola Benes, Jan Kretínský, Kim Guldstrand Larsen, and Jirí Srba. On determinism in modal transition systems. *Theoretical Computer Science*, 410(41):4026–4043, 2009.

[27] Benoît Caillaud and Jean-Baptiste Raclet. Ensuring Reachability by Design. In *Int. Colloquium on Theoretical Aspects of Computing*, sept 2012.

[28] Albert Benveniste and Gérard Berry. The synchronous approach to reactive and real-time systems. *Proceedings of the IEEE*, 79(9):1270–1282, 1991.

[29] Albert Benveniste, Benoît Caillaud, Alberto Ferrari, Leonardo Mangeruca, Roberto Passerone, and Christos Sofronis. Multiple viewpoint contract-based specification and design. In *Proceedings of the Software Technology Concertation on Formal Methods for Components and Objects, FMCO'07*, volume 5382 of *Lecture Notes in Computer Science*, pages 200–225. Springer, October 2008.

[30] Albert Benveniste, Paul Caspi, Stephen A. Edwards, Nicolas Halbwachs, Paul Le Guernic, and Robert de Simone. The Synchronous Languages 12 years later. *Proceedings of the IEEE*, 91(1):64–83, 2003.

[31] Luca Benvenuti, Alberto Ferrari, Leonardo Mangeruca, Emanuele Mazzi, Roberto Passerone, and Christos Sofronis. A contract-based formalism for the specification of heterogeneous systems. In *Proceedings of the Forum on Specification, Verification and Design Languages (FDL08)*, pages 142–147, Stuttgart, Germany, September 23–25, 2008.

[32] Gerard Berry. The effectiveness of synchronous languages for the development of safety-critical systems. White paper, Esterel Technologies, 2003.

[33] Nathalie Bertrand, Axel Legay, Sophie Pinchinat, and Jean-Baptiste Raclet. A compositional approach on modal specifications for timed systems. In *Proc. of the 11th International Conference on Formal Engineering Methods (ICFEM'09)*, volume 5885 of *Lecture Notes in Computer Science*, pages 679–697. Springer, 2009.

[34] Nathalie Bertrand, Axel Legay, Sophie Pinchinat, and Jean-Baptiste Raclet. Modal event-clock specifications for timed component-based design. *Science of Computer Programming*, 2011. To appear.

[35] Nathalie Bertrand, Axel Legay, Sophie Pinchinat, and Jean-Baptiste Raclet. Modal event-clock specifications for timed component-based design. *Science of Computer Programming*, 2012. to appear.

[36] Nathalie Bertrand, Sophie Pinchinat, and Jean-Baptiste Raclet. Refinement and consistency of timed modal specifications. In *Proc. of the 3rd International Conference on Language and Automata Theory and Applications (LATA'09)*, volume 5457 of *Lecture Notes in Computer Science*, pages 152–163. Springer, 2009.

[37] Antoine Beugnard, Jean-Marc Jézéquel, and Noël Plouzeau. Making components contract aware. *IEEE Computer*, 32(7):38–45, 1999.

[38] Dirk Beyer, Arindam Chakrabarti, and Thomas A. Henzinger. Web service interfaces. In Allan Ellis and Tatsuya Hagino, editors, *WWW*, pages 148–159. ACM, 2005.

[39] Purandar Bhaduri and S. Ramesh. Interface synthesis and protocol conversion. *Formal Aspects of Computing*, 20(2):205–224, 2008.

[40] Purandar Bhaduri and Ingo Stierand. A proposal for real-time interfaces in speeds. In *Design, Automation and Test in Europe (DATE'10)*, pages 441–446. IEEE, 2010.

[41] Simon Bliudze and Joseph Sifakis. The Algebra of Connectors - Structuring Interaction in BIP. *IEEE Trans. Computers*, 57(10):1315–1330, 2008.

[42] R. Bloem and B. Jobstmann. Manual for property-based synthesis tool. Technical Report Prosyd D2.2/3, 2006.

[43] Roderick Bloem, Alessandro Cimatti, Karin Greimel, Georg Hofferek, Robert Könighofer, Marco Roveri, Viktor Schuppan, and Richard Seeber. RATSY - A New Requirements Analysis Tool with Synthesis. In Tayssir Touili, Byron Cook, and Paul Jackson, editors, *CAV*, volume 6174 of *Lecture Notes in Computer Science*, pages 425–429. Springer, 2010.

[44] Grady Booch, James Rumbaugh, and Ivar Jacobson. *Unified Modeling Language User Guide, The (2nd Edition) (Addison-Wesley Object Technology Series)*. Addison-Wesley Professional, 2005.

[45] Amar Bouali. XEVE, an Esterel Verification Environment. In Alan J. Hu and Moshe Y. Vardi, editors, *CAV*, volume 1427 of *Lecture Notes in Computer Science*, pages 500–504. Springer, 1998.

[46] Gérard Boudol and Kim Guldstrand Larsen. Graphical versus logical specifications. *Theor. Comput. Sci.*, 106(1):3–20, 1992.

[47] Jerry R. Burch. *Trace Algebra for Automatic Verification of Real-Time Concurrent Systems*. PhD thesis, School of Computer Science, Carnegie Mellon University, August 1992.

[48] Jerry R. Burch, Roberto Passerone, and Alberto L. Sangiovanni-Vincentelli. Overcoming heterophobia: Modeling concurrency in heterogeneous systems. In *Proceedings of the 2^{nd} International Conference on Application of Concurrency to System Design (ACSD01)*, pages 13–32, Newcastle upon Tyne, UK, June 25–29, 2001. IEEE Computer Society, Los Alamitos, CA, USA.

[49] Benoît Caillaud. Mica: A Modal Interface Compositional Analysis Library, October 2011. http://www.irisa.fr/s4/tools/mica.

[50] Benoît Caillaud, Benoît Delahaye, Kim G. Larsen, Axel Legay, Mikkel Larsen Pedersen, and Andrzej Wasowski. Compositional design methodology with constraint Markov chains. In *Proceedings of the 7th International Conference on Quantitative Evaluation of SysTems (QEST) 2010*. IEEE Computer Society, 2010.

[51] Daniela Cancila, Roberto Passerone, Tullio Vardanega, and Marco Panunzio. Toward correctness in the specification and handling of non-functional attributes of high-integrity real-time embedded systems. *IEEE Transactions on Industrial Informatics*, 6(2):181–194, May 2010.

[52] Karlis Cerans, Jens Chr. Godskesen, and Kim Guldstrand Larsen. Timed Modal Specification - Theory and Tools. In Costas Courcoubetis, editor, *CAV*, volume 697 of *Lecture Notes in Computer Science*, pages 253–267. Springer, 1993.

[53] Pavol Cerný, Martin Chmelik, Thomas A. Henzinger, and Arjun Radhakrishna. Interface simulation distances. In Marco Faella and Aniello Murano, editors, *GandALF*, volume 96 of *EPTCS*, pages 29–42, 2012.

[54] Arindam Chakrabarti. *A Framework for Compositional Design and Analysis of Systems*. PhD thesis, EECS Department, University of California, Berkeley, Dec 2007.

[55] Arindam Chakrabarti, Luca de Alfaro, Thomas A. Henzinger, Marcin Jurdzinski, and Freddy Y. C. Mang. Interface compatibility checking for software modules. In Ed Brinksma and Kim Guldstrand Larsen, editors, *CAV*, volume 2404 of *Lecture Notes in Computer Science*, pages 428–441. Springer, 2002.

[56] Arindam Chakrabarti, Luca de Alfaro, Thomas A. Henzinger, and Freddy Y. C. Mang. Synchronous and Bidirectional Component Interfaces. In *Proc. of the 14th International Conference on Computer Aided Verification (CAV'02)*, volume 2404 of *Lecture Notes in Computer Science*, pages 414–427. Springer, 2002.

[57] Arindam Chakrabarti, Luca de Alfaro, Thomas A. Henzinger, and Mariëlle Stoelinga. Resource Interfaces. In Rajeev Alur and Insup Lee, editors, *EMSOFT*, volume 2855 of *Lecture Notes in Computer Science*, pages 117–133. Springer, 2003.

[58] Edward Y. Chang, Zohar Manna, and Amir Pnueli. Characterization of temporal property classes. In Werner Kuich, editor, *ICALP*, volume 623 of *Lecture Notes in Computer Science*, pages 474–486. Springer, 1992.

[59] Taolue Chen, Chris Chilton, Bengt Jonsson, and Marta Z. Kwiatkowska. A compositional specification theory for component behaviours. In Helmut Seidl, editor, *ESOP*, volume 7211 of *Lecture Notes in Computer Science*, pages 148–168. Springer, 2012.

[60] Chris Chilton, Marta Z. Kwiatkowska, and Xu Wang. Revisiting timed specification theories: A linear-time perspective. In Marcin Jurdzinski and Dejan Nickovic, editors, *FORMATS*, volume 7595 of *Lecture Notes in Computer Science*, pages 75–90. Springer, 2012.

[61] E. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 1999.

[62] Edmund M. Clarke, David E. Long, and Kenneth L. McMillan. Compositional model checking. In *LICS*, pages 353–362, 1989.

[63] W. Damm, E. Thaden, I. Stierand, T. Peikenkamp, and H. Hungar. Using Contract-Based Component Specifications for Virtual Integration and Architecture Design. In *Proceedings of the 2011 Design, Automation and Test in Europe (DATE'11)*, March 2011. To appear.

[64] Werner Damm and David Harel. LSCs: Breathing life into message sequence charts. *Formal Methods in System Design*, 19(1):45–80, 2001.

[65] Werner Damm, Angelika Votintseva, Alexander Metzner, Bernhard Josko, Thomas Peikenkamp, and Eckard Böde. Boosting reuse of embedded automotive applications through rich components. In *Proceedings of FIT 2005 - Foundations of Interface Technologies*, 2005.

[66] Werner Damm and Bernd Westphal. Live and let die: LSC based verification of UML models. *Sci. Comput. Program.*, 55(1-3):117–159, 2005.

[67] Abhijit Davare, Douglas Densmore, Trevor Meyerowitz, Alessandro Pinto, Alberto Sangiovanni-Vincentelli, Guang Yang, and Qi Zhu. A next-generation design framework for platform-based design. In *Design Verification Conference (DVCon)*, San Jose', California, 2007.

[68] Alexandre David, Kim. G. Larsen, Axel Legay, Ulrik Nyman, and Andrzej Wasowski. Timed I/O automata : A complete specification theory for real-time systems. In *Proc. of the 13th ACM International Conference on Hybrid Systems: Computation and Control (HSCC'10)*, pages 91–100. ACM, 2010.

[69] Alexandre David, Kim Guldstrand Larsen, Axel Legay, Ulrik Nyman, and Andrzej Wasowski. ECDAR: An Environment for Compositional Design and Analysis of Real Time Systems. In *Proc. of the 8th International Symposium on Automated Technology for Verification and Analysis (ATVA'10)*, volume 6252 of *Lecture Notes in Computer Science*, pages 365–370, 2010.

[70] Luca de Alfaro. Game Models for Open Systems. In *Verification: Theory and Practice*, volume 2772 of *Lecture Notes in Computer Science*, pages 269–289. Springer, 2003.

[71] Luca de Alfaro, Leandro Dias da Silva, Marco Faella, Axel Legay, Pritam Roy, and Maria Sorea. Sociable Interfaces. In *Proc. of*

the 5th International Workshop on Frontiers of Combining Systems (FroCos'05), volume 3717 of *Lecture Notes in Computer Science*, pages 81–105. Springer, 2005.

[72] Luca de Alfaro and Thomas A. Henzinger. Interface automata. In *Proc. of the 9th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE'01)*, pages 109–120. ACM Press, 2001.

[73] Luca de Alfaro and Thomas A. Henzinger. Interface theories for component-based design. In Thomas A. Henzinger and Christoph M. Kirsch, editors, *EMSOFT*, volume 2211 of *Lecture Notes in Computer Science*, pages 148–165. Springer, 2001.

[74] Luca de Alfaro, Thomas A. Henzinger, and Mariëlle Stoelinga. Timed Interfaces. In *Proc. of the 2nd International Workshop on Embedded Software (EMSOFT'02)*, volume 2491 of *Lecture Notes in Computer Science*, pages 108–122. Springer, 2002.

[75] Benoît Delahaye. *Modular Specification and Compositional Analysis of Stochastic Systems*. PhD thesis, Université de Rennes 1, 2010.

[76] Benoît Delahaye, Benoît Caillaud, and Axel Legay. Probabilistic Contracts : A Compositional Reasoning Methodology for the Design of Stochastic Systems. In *Proc. 10th International Conference on Application of Concurrency to System Design (ACSD), Braga, Portugal*. IEEE, 2010.

[77] Benoît Delahaye, Benoît Caillaud, and Axel Legay. Probabilistic contracts : A compositional reasoning methodology for the design of systems with stochastic and/or non-deterministic aspects. *Formal Methods in System Design*, 2011. To appear.

[78] Benoît Delahaye, Uli Fahrenberg, Thomas A. Henzinger, Axel Legay, and Dejan Nickovic. Synchronous interface theories and time triggered scheduling. In Holger Giese and Grigore Rosu, editors, *FMOODS/FORTE*, volume 7273 of *Lecture Notes in Computer Science*, pages 203–218. Springer, 2012.

[79] Benoît Delahaye, Joost-Pieter Katoen, Kim G. Larsen, Axel Legay, Mikkel L. Pedersen, Falak Sher, and Andrzej Wasowski. Abstract Probabilistic Automata. In Ranjit Jhala and David A. Schmidt, editors, *VMCAI*, volume 6538 of *Lecture Notes in Computer Science*, pages 324–339. Springer, 2011.

[80] Douglas Densmore, Sanjay Rekhi, and Alberto L. Sangiovanni-Vincentelli. Microarchitecture Development via Metropolis Successive Platform Refinement. In *DATE*, pages 346–351. IEEE Computer Society, 2004.

[81] David L. Dill. *Trace Theory for Automatic Hierarchical Verification of Speed-Independent Circuits*. ACM Distinguished Dissertations. MIT Press, 1989.

[82] Nicolás D'Ippolito, Dario Fischbein, Marsha Chechik, and Sebastián Uchitel. MTSA: The Modal Transition System Analyser. In *Proc. of the 23rd IEEE/ACM International Conference on Automated Software Engineering (ASE'08)*, pages 475–476. IEEE, 2008.

[83] Laurent Doyen, Thomas A. Henzinger, Barbara Jobstmann, and Tatjana Petrov. Interface theories with component reuse. In *Proceedings of the 8th ACM & IEEE International conference on Embedded software, EMSOFT'08*, pages 79–88, 2008.

[84] Dumitru Potop-Butucaru and Stephen Edwards and Gérard Berry. *Compiling Esterel*. Springer V., 2007. ISBN: 0387706267.

[85] Cindy Eisner. PSL for Runtime Verification: Theory and Practice. In Oleg Sokolsky and Serdar Tasiran, editors, *RV*, volume 4839 of *Lecture Notes in Computer Science*, pages 1–8. Springer, 2007.

[86] Cindy Eisner, Dana Fisman, John Havlicek, Michael J.C. Gordon, Anthony McIsaac, and David Van Campenhout. Formal Syntax and Semantics of PSL - Appendix B of Accellera LRM January 2003. Technical report, IBM, 2003.

[87] Cindy Eisner, Dana Fisman, John Havlicek, Yoad Lustig, Anthony McIsaac, and David Van Campenhout. Reasoning with temporal logic on truncated paths. In Warren A. Hunt Jr. and Fabio Somenzi, editors, *CAV*, volume 2725 of *Lecture Notes in Computer Science*, pages 27–39. Springer, 2003.

[88] Johan Eker, Jörn W. Janneck, Edward A. Lee, Jie Liu, Xiaojun Liu, J. Ludvig, Stephen Neuendorffer, S. Sachs, and Yuhong Xiong. Taming heterogeneity - the ptolemy approach. *Proc. of the IEEE*, 91(1):127–144, 2003.

[89] G. Feuillade. Modal specifications are a syntactic fragment of the Mu-calculus. Research Report RR-5612, INRIA, June 2005.

[90] Dario Fischbein and Sebastián Uchitel. On correct and complete strong merging of partial behaviour models. In *Proc. of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering (SIGSOFT FSE'08)*, pages 297–307. ACM, 2008.

[91] F. Fleurey, P. A. Muller, and J. M. Jzquel. Weaving executability into object-oriented meta-languages. In *Proceedgins of the 8^{th} International Conference on Model Driven Engineering Languages and Systems (MODELS05)*, October 2005.

[92] Frédéric Boussinot and Robert de Simone. The Esterel language. *Proceedings of the IEEE*, 79(9):1293–1304, 1991.

[93] Peter Fritzson. *Principles of Object-Oriented Modeling and Simulation with Modelica 2.1*. Wiley, 2003.

[94] Tayfun Gezgin, Raphael Weber, and Maurice Girod. A Refinement Checking Technique for Contract-Based Architecture Designs. In *Fourth International Workshop on Model Based Architecting and Construction of Embedded Systems, ACES-MB'11*, volume 7167 of *Lecture Notes in Computer Science*. Springer, October 2011.

[95] Jens Chr. Godskesen, Kim Guldstrand Larsen, and Arne Skou. Automatic verification of real-time systems using Epsilon. In Son T. Vuong and Samuel T. Chanson, editors, *PSTV*, volume 1 of *IFIP Conference Proceedings*, pages 323–330. Chapman & Hall, 1994.

[96] G. Gössler and J.-B. Raclet. Modal Contracts for Component-based Design. In *Proc. of the 7th IEEE International Conference on Software Engineering and Formal Methods (SEFM'09)*. IEEE Computer Society Press, November 2009.

[97] Susanne Graf and Sophie Quinton. Contracts for BIP: Hierarchical Interaction Models for Compositional Verification. In John Derrick and Jüri Vain, editors, *FORTE*, volume 4574 of *Lecture Notes in Computer Science*, pages 1–18. Springer, 2007.

[98] Orna Grumberg and David E. Long. Model checking and modular verification. *ACM Trans. Program. Lang. Syst.*, 16(3):843–871, 1994.

[99] Imene Ben Hafaiedh, Susanne Graf, and Sophie Quinton. Reasoning about Safety and Progress Using Contracts. In *Proc. of ICFEM'10*, volume 6447 of *LNCS*, pages 436–451. Springer, 2010.

[100] Nicolas Halbwachs. *Synchronous programming of reactive systems*. Kluwer Academic Pub., 1993.

[101] Nicolas Halbwachs, Fabienne Lagnier, and Christophe Ratel. Programming and Verifying Real-Time Systems by Means of the Synchronous Data-Flow Language Lustre. *IEEE Trans. Software Eng.*, 18(9):785–793, 1992.

[102] Nicolas Halbwachs, Fabienne Lagnier, and Pascal Raymond. Synchronous observers and the verification of reactive systems. In Maurice Nivat, Charles Rattray, Teodor Rus, and Giuseppe Scollo, editors, *AMAST*, Workshops in Computing, pages 83–96. Springer, 1993.

[103] Nicolas Halbwachs and Pascal Raymond. Validation of synchronous reactive systems: From formal verification to automatic testing. In P. S. Thiagarajan and Roland H. C. Yap, editors, *ASIAN*, volume 1742 of *Lecture Notes in Computer Science*, pages 1–12. Springer, 1999.

[104] David Harel. Statecharts: A visual formalism for complex systems. *Sci. Comput. Program.*, 8(3):231–274, 1987.

[105] David Harel, Hillel Kugler, Shahar Maoz, and Itai Segall. Accelerating smart play-out. In Jan van Leeuwen, Anca Muscholl, David Peleg, Jaroslav Pokorný, and Bernhard Rumpe, editors, *SOFSEM*, volume 5901 of *Lecture Notes in Computer Science*, pages 477–488. Springer, 2010.

[106] David Harel, Robby Lampert, Assaf Marron, and Gera Weiss. Model-checking behavioral programs. In Samarjit Chakraborty, Ahmed Jerraya, Sanjoy K. Baruah, and Sebastian Fischmeister, editors, *EMSOFT*, pages 279–288. ACM, 2011.

[107] David Harel and Rami Marelly. *Come, Let's Play: Scenario-Based Programming Using LSCs and the Play-Engine*. Springer-Verlag, 2003. http://www.wisdom.weizmann.ac.il/~harel/ComeLetsPlay.pdf.

[108] David Harel, Assaf Marron, and Gera Weiss. Behavioral programming. *Commun. ACM*, 55(7):90–100, 2012.

[109] David Harel, Assaf Marron, Guy Wiener, and Gera Weiss. Behavioral programming, decentralized control, and multiple time scales. In Cristina Videira Lopes, editor, *SPLASH Workshops*, pages 171–182. ACM, 2011.

[110] David Harel and Amir Pnueli. On the development of reactive systems. In K. R. Apt, editor, *Logic and Models for Verification and Specification of Concurrent Systems*, volume F13 of *NATO ASI Series*, pages 477–498. Springer-Verlag, 1985.

[111] H. Heinecke, W. Damm, B. Josko, A. Metzner, H. Kopetz, A. Sangiovanni-Vincentelli, and M. Di Natale. Software Components for Reliable Automotive Systems. In *Design, Automation and Test in Europe, 2008. DATE '08*, pages 549–554, march 2008.

[112] Thomas A. Henzinger and Dejan Nickovic. Independent implementability of viewpoints. In Radu Calinescu and David Garlan, edi-

tors, *Monterey Workshop*, volume 7539 of *Lecture Notes in Computer Science*, pages 380–395. Springer, 2012.

[113] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, 1969.

[114] INCOSE. Incose systems engineering handbook, 2010. http://www.incose.org/ProductsPubs/products/sehandbook.aspx.

[115] Cliff B. Jones. Specification and design of (parallel) programs. In *IFIP Congress*, pages 321–332, 1983.

[116] B. Jonsson and K. G. Larsen. Specification and refinement of probabilistic processes. In *Logic in Computer Science (LICS)*, pages 266–277. IEEE Computer, 1991.

[117] Gilles Kahn. The Semantics of Simple Language for Parallel Programming. In *IFIP Congress*, pages 471–475, 1974.

[118] S. Karris. *Introduction to Simulink with Engineering Applications*. Orchard Publications, 2006.

[119] G. Karsai, J. Sztipanovits, A. Ledeczi, and T. Bapty. Model-integrated development of embedded software. *Proceedings of the IEEE*, 91(1):145–164, January 2003.

[120] Gabor Karsai, Janos Sztipanovitz, Akos Ledczi, and Ted Bapty. Model-integrated development of embedded software. *Proceedings of the IEEE*, 91(1), January 2003.

[121] Orna Kupferman and Moshe Y. Vardi. Modular model checking. In Willem P. de Roever, Hans Langmaack, and Amir Pnueli, editors, *COMPOS*, volume 1536 of *Lecture Notes in Computer Science*, pages 381–401. Springer, 1997.

[122] Leslie Lamport. Proving the correctness of multiprocess programs. *IEEE Trans. Software Eng.*, 3(2):125–143, 1977.

[123] C. Larman and V.R. Basili. Iterative and incremental developments: a brief history. *Computer*, 36(6):47–56, June 2003.

[124] Kim G. Larsen and L. Xinxin. Equation solving using modal transition systems. In *Proceedings of the 5th Annual IEEE Symp. on Logic in Computer Science, LICS'90*, pages 108–117. IEEE Computer Society Press, 1990.

[125] Kim Guldstrand Larsen. Modal specifications. In *Automatic Verification Methods for Finite State Systems*, volume 407 of *Lecture Notes in Computer Science*, pages 232–246. Springer, 1989.

[126] Kim Guldstrand Larsen, Ulrik Nyman, and Andrzej Wasowski. Interface Input/Output Automata. In Jayadev Misra, Tobias Nipkow, and Emil Sekerinski, editors, *FM*, volume 4085 of *Lecture Notes in Computer Science*, pages 82–97. Springer, 2006.

[127] Kim Guldstrand Larsen, Ulrik Nyman, and Andrzej Wasowski. Modal I/O Automata for Interface and Product Line Theories. In *Programming Languages and Systems, 16th European Symposium on Programming, ESOP'07*, volume 4421 of *Lecture Notes in Computer Science*, pages 64–79. Springer, 2007.

[128] Kim Guldstrand Larsen, Ulrik Nyman, and Andrzej Wasowski. On Modal Refinement and Consistency. In *Proc. of the 18th International Conference on Concurrency Theory (CONCUR'07)*, pages 105–119. Springer, 2007.

[129] Kim Guldstrand Larsen, Bernhard Steffen, and Carsten Weise. A constraint oriented proof methodology based on modal transition systems. In *Proc. of the 1st International Workshop on Tools and Algorithms for Construction and Analysis of Systems (TACAS'95)*, volume 1019 of *Lecture Notes in Computer Science*, pages 17–40. Springer, 1995.

[130] Kim Guldstrand Larsen and Bent Thomsen. A Modal Process Logic. In *Proceedings of the Third Annual Symposium on Logic in Computer Science (LICS'88)*, pages 203–210. IEEE, 1988.

[131] A. Ledeczi, A. Bakay, M. Maroti, P. Volgyesi, G. Nordstrom, J. Sprinkle, and G. Karsai. Composing domain-specific design environments. *IEEE Computer*, 34(11):44 –51, November 2001.

[132] Akos Ledeczi, Miklos Maroti, Arpad Bakay, Gabor Karsai, Jason Garrett, Charles Thomason, Greg Nordstrom, Jonathan Sprinkle, and Peter Volgyesi. The generic modeling environment. In *Proceedings of the IEEE International Workshop on Intelligent Signal Processing (WISP2001)*, Budapest, Hungary, May 24–25 2001.

[133] Edward A. Lee. Cyber physical systems: Design challenges. Technical Report UCB/EECS-2008-8, EECS Department, University of California, Berkeley, Jan 2008.

[134] Nancy A. Lynch. Input/output automata: Basic, timed, hybrid, probabilistic, dynamic, .. In Roberto M. Amadio and Denis Lugiez, editors, *CONCUR*, volume 2761 of *Lecture Notes in Computer Science*, pages 187–188. Springer, 2003.

[135] Nancy A. Lynch and Eugene W. Stark. A proof of the kahn principle for input/output automata. *Inf. Comput.*, 82(1):81–92, 1989.

[136] Zohar Manna and Amir Pnueli. *Temporal verification of reactive systems: Safety*. Springer, 1995.

[137] Hervé Marchand, Patricia Bournai, Michel Le Borgne, and Paul Le Guernic. Synthesis of Discrete-Event Controllers Based on the Signal Environment. *Discrete Event Dynamic Systems*, 10(4):325–346, 2000.

[138] Hervé Marchand and Mazen Samaan. Incremental Design of a Power Transformer Station Controller Using a Controller Synthesis Methodology. *IEEE Trans. Software Eng.*, 26(8):729–741, 2000.

[139] B. Meyer. Applying "design by contract". *IEEE Computer*, 25(10):40–51, October 1992.

[140] Bertrand Meyer. *Touch of Class: Learning to Program Well Using Object Technology and Design by Contract*. Springer, Software Engineering, 2009.

[141] M.W. Maier. Architecting Principles for Systems of Systems. *Systems Engineering*, 1(4):267–284, 1998.

[142] Walid A. Najjar, Edward A. Lee, and Guang R. Gao. Advances in the dataflow computational model. *Parallel Computing*, 25(13-14):1907–1929, 1999.

[143] Radu Negulescu. *Process Spaces and the Formal Verification of Asynchronous Circuits*. PhD thesis, University of Waterloo, Canada, 1998.

[144] Nicolas Halbwachs and Paul Caspi and Pascal Raymond and Daniel Pilaud. The synchronous data flow programming language Lustre. *Proceedings of the IEEE*, 79(9):1305–1320, 1991.

[145] P. Nuzzo, A. Sangiovanni-Vincentelli, X. Sun, and A. Puggelli. Methodology for the design of analog integrated interfaces using contracts. *IEEE Sensors Journal*, 12(12):3329–3345, Dec. 2012.

[146] Object Management Group (OMG). Model driven architecture (MDA) FAQ. [online], http://www.omg.org/mda/.

[147] Object Management Group (OMG). Unified Modeling Language (UML) specification. [online], http://www.omg.org/spec/UML/.

[148] Object Management Group (OMG), . A UML profile for MARTE, beta 1. OMG Adopted Specification ptc/07-08-04, OMG, August 2007.

[149] Object Management Group (OMG), . System modeling language specification v1.1. Technical report, OMG, 2008.

[150] Object constraint language, version 2.0. OMG Available Specification formal/06-05-01, Object Management Group, May 2006.

[151] The Design Automation Standards Committee of the IEEE Computer Society, editor. *1850-2010 - IEEE Standard for Property Specification Language (PSL)*. IEEE Computer Society, 2010.

[152] J. Hudak P. Feiler, D. Gluch. The Architecture Analysis and Design Language (AADL): An Introduction. *Software Engineering Institute (SEI) Technical Note, CMU/SEI-2006-TN-011*, February 2006.

[153] Roberto Passerone. *Semantic Foundations for Heterogeneous Systems*. PhD thesis, Department of Electrical Engineering and Computer Sciences, University of California, Berkeley, Berkeley, CA 94720, May 2004.

[154] Roberto Passerone, Jerry R. Burch, and Alberto L. Sangiovanni-Vincentelli. Refinement preserving approximations for the design and verification of heterogeneous systems. *Formal Methods in System Design*, 31(1):1–33, August 2007.

[155] Roberto Passerone, Luca de Alfaro, Thomas A. Henzinger, and Alberto Sangiovanni-Vincentelli. Convertibility verification and converter synthesis: Two faces of the same coin. In *Proceedings of International Conference on Computer Aided Design*, San Jose, CA., 2002.

[156] Roberto Passerone, Imene Ben Hafaiedh, Susanne Graf, Albert Benveniste, Daniela Cancila, Arnaud Cuccuru, Sébastien Gérard, Francois Terrier, Werner Damm, Alberto Ferrari, Leonardo Mangeruca, Bernhard Josko, Thomas Peikenkamp, and Alberto Sangiovanni-Vincentelli. Metamodels in Europe: Languages, tools, and applications. *IEEE Design and Test of Computers*, 26(3):38–53, May/June 2009.

[157] Paul Le Guernic and Thiserry Gautier and Michel Le Borgne and Claude Le Maire. Programming real-time applications with Signal. *Proceedings of the IEEE*, 79(9):1321–1336, 1991.

[158] I. Pill, B. Jobstmann, R. Bloem, R. Frank, M. Moulin, B. Sterin, M. Roveri, and S. Semprini. Property simulation. Technical Report Prosyd D1.2/1, 2005.

[159] Dumitru Potop-Butucaru, Benoît Caillaud, and Albert Benveniste. Concurrency in synchronous systems. *Formal Methods in System Design*, 28(2):111–130, 2006.

[160] Dumitru Potop-Butucaru, Robert de Simone, and Yves Sorel. Necessary and sufficient conditions for deterministic desynchronization. In Christoph M. Kirsch and Reinhard Wilhelm, editors, *EMSOFT*, pages 124–133. ACM, 2007.

[161] Terry Quatrani. *Visual modeling with Rational Rose 2000 and UML (2nd ed.)*. Addison-Wesley Longman Ltd., Essex, UK, UK, 2000.

[162] R. Sudarsan and S.J. Fenves and R.D. Sriram and F. Wang. A product information modeling framework for product lifecycle management. *Computer-Aided Design*, 37:1399–1411, 2005.

[163] Jean-Baptiste Raclet. *Quotient de spécifications pour la réutilisation de composants*. PhD thesis, Ecole doctorale Matisse, université de Rennes 1, November 2007.

[164] Jean-Baptiste Raclet. Residual for Component Specifications. In *Proc. of the 4th International Workshop on Formal Aspects of Component Software (FACS'07)*, 2007.

[165] Jean-Baptiste Raclet, Eric Badouel, Albert Benveniste, Benoît Caillaud, Axel Legay, and Roberto Passerone. Modal interfaces: Unifying interface automata and modal specifications. In *Proceedings of the Ninth International Conference on Embedded Software (EMSOFT09)*, pages 87–96, Grenoble, France, October 12–16, 2009.

[166] Jean-Baptiste Raclet, Eric Badouel, Albert Benveniste, Benoît Caillaud, Axel Legay, and Roberto Passerone. A modal interface theory for component-based design. *Fundamenta Informaticae*, 108(1-2):119–149, 2011.

[167] Jean-Baptiste Raclet, Eric Badouel, Albert Benveniste, Benoît Caillaud, and Roberto Passerone. Why are modalities good for interface theories? In *Proc. of the 9th International Conference on Application of Concurrency to System Design (ACSD'09)*. IEEE Computer Society Press, 2009.

[168] Richard Payne and John Fitzgerald. Evaluation of Architectural Frameworks Supporting Contract-Based Specification. Technical Report CS-TR-1233, Computing Science, Newcastle University, UK, Dec 2010. available from http://www.cs.ncl.ac.uk/publications/trs/papers/1233.pdf.

[169] Robert W. Floyd. Assigning meaning to programs. In J.T. Schwartz, editor, *Proceedings of Symposium on Applied Mathematics*, volume 19, pages 19–32, 1967.

[170] A. Sangiovanni-Vincentelli, S. Shukla, J. Sztipanovits, G. Yang, and D. Mathaikutty. Metamodeling: An emerging representation paradigm for system-level design". *Special Section on Meta-Modeling, IEEE Design and Test,*, 26(3):54–69, 2009.

[171] Alberto Sangiovanni-Vincentelli. Quo vadis, sld?: Reasoning about the trends and challenges of system level design. *Proc. of the IEEE*, 95(3):467–506, 2007.

[172] D. Schmidt. Model-driven engineering. *IEEE Computer*, pages 25–31, February 2006.

[173] German Sibay, Sebastian Uchitel, and Víctor Braberman. Existential Live Sequence Charts Revisited,. In *ICSE 2008: 30th International Conference on Software Engineering*. ACM, May 2008.

[174] Joseph Sifakis. Component-Based Construction of Heterogeneous Real-Time Systems in Bip. In Giuliana Franceschinis and Karsten Wolf, editors, *Petri Nets*, volume 5606 of *Lecture Notes in Computer Science*, page 1. Springer, 2009.

[175] Functional safety of electrical/electronic/programmable electronic safety-related systems. Standard IEC 61508.

[176] Eugene W. Stark. A proof technique for rely/guarantee properties. In S. N. Maheshwari, editor, *FSTTCS*, volume 206 of *Lecture Notes in Computer Science*, pages 369–391. Springer, 1985.

[177] Stavros Tripakis, Ben Lickly, Thomas A. Henzinger, and Edward A. Lee. On relational interfaces. In *Proc. of the 9th ACM & IEEE International conference on Embedded software (EMSOFT'09)*, pages 67–76. ACM, 2009.

[178] Stavros Tripakis, Ben Lickly, Thomas A. Henzinger, and Edward A. Lee. A theory of synchronous relational interfaces. *ACM Trans. Program. Lang. Syst.*, 33(4):14, 2011.

[179] Sebastián Uchitel and Marsha Chechik. Merging partial behavioural models. In *Proc. of the 12th ACM SIGSOFT International Symposium on Foundations of Software Engineering (SIGSOFT FSE'10)*, pages 43–52. ACM, 2004.

[180] Jos Warmer and Anneke Kleppe. *The Object Constraint Language: Getting Your Models Ready for MDA*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2nd edition, 2003.

RESEARCH CENTRE
RENNES – BRETAGNE ATLANTIQUE

Campus universitaire de Beaulieu
35042 Rennes Cedex

Publisher
Inria
Domaine de Voluceau - Rocquencourt
BP 105 - 78153 Le Chesnay Cedex
inria.fr

ISSN 0249-6399