Contracts for Systems Design: Theory

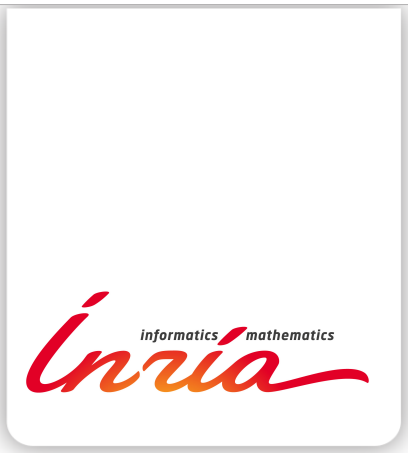
Albert Benveniste, Benoît Caillaud, Dejan Nickovic, Roberto Passerone, Jean-Baptiste Raclet, Philipp Reinkemeier, Alberto Sangiovanni-Vincentelli, Werner Damm, Tom Henzinger, Kim Guldstrand Larsen

Contracts for Systems Design: Theory

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

RESEARCH REPORT
N° 8759
July 2015
Project-Teams Hycomes

ISSN 0249-6399 ISRN INRIA/RR--8759--FR+ENG



Contracts for Systems Design: Theory

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

Project-Teams Hycomes

Research Report n° 8759 — July 2015 — 86 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 TerraSwarm project funded by the STARnet phase of the Focus Center Research Program (FCRP) administered by the Semiconductor Research Corporation (SRC), the iCyPhy program sponsored by IBM and United Technology Corporation, the VKR Center of Excellence MT-LAB, and the German Innovation Alliance on Embedded Systems SPES2020.

Albert Benveniste and Benoît Caillaud are with INRIA, Rennes, France; Dejan Nickovic is with Austrian Institute of Technology (AIT), Roberto Passerone is with University of Trento, Italy, Jean-Baptiste Raclet is with IRIT-CNRS, Toulouse, France, Philipp Reinkemeier and Werner Damm are with Offis and University of Oldenburg, Alberto Sangiovanni-Vincentelli is with University of California at Berkeley, USA, Tom Henzinger is with IST Austria, Klosterneuburg, Kim G. Larsen is with Aalborg University, Danmark

* INRIA, Rennes, France. corresp. author: Albert.Benveniste@inria.fr

RESEARCH CENTRE
RENNES – BRETAGNE ATLANTIQUE

Campus universitaire de Beaulieu
35042 Rennes Cedex

Abstract: 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.
Recently an "orthogonal" approach has been proposed that can be applied to all methodologies introduced 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 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 interface and contract theories to show similarities and correspondences.
This paper is complemented by a companion paper [30] where contract based design is illustrated through use cases.
These results were announced in the report [29]

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

Contrats pour la conception de systèmes: théorie

Résumé : La conception de système constitue une étape clé pour la conception des avions, des trains, des voitures, etc. La complexité croissante des ces systèmes, largement due au logiciel, est source de retards et dépassements de coût. Les ”bonnes pratiques” ne suffisent pas à régler ce problème et de nouvelles approches sont nécessaires. La conception fondée sur des modèles, complétée par la conception par niveaux et par composants, constituent un premier progrès. Récemment, une approche originale a été proposée, qui peut s'appliquer à toutes les méthodologies ci-dessus: la *conception par contrats.* De nombreux résultats existent dans ce domaine mais il manquait une vision unifiée qui mette en perspective des approches apparemment différentes telles que les contrats hypothèse/garantie ou les interfaces. Cet article a pour ambition d'apporter une telle vision unifiée.

Cet article est complété par l'article [30] où la conception par contrats est illustrée sur des cas d'application.

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

Contents

1 Introduction: why contract based 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. Many challenges face the system community to deliver products that are reliable and effective.

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

Table 1: Difficulties related to system integration. The table displays, for each industrial sector, the percentage of tasks delayed and tasks causing delays, for the different phases of system design.

The first issue is the complexity of systems, regarding both architecture alternatives, the embedded software, the underlying platform of predefined components, and system integration. Table 1[1] displays the share of these different items in the difficulties related to systems complexity. This table highlights the importance of system integration, where corrections occur late in the design flow and are therefore very costly.

System integration is particularly critical for OEMs managing the integration and maintenance process with subsystems that come from different suppliers who use different design methods, different software architectures, and different hardware plat-

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

forms. Technical annexes to commercial contracts between OEM and suppliers are the first source of problems. Specifications used for procurement should be precise, unambiguous, and complete. Indeed, a recurrent reason for failures causing deep iterations across supply chain boundaries rests in incomplete characterizations of the conditions for use and environment of the system to be developed by the supplier, such as missing information about failure modes and failure rates, missing information on possible sources of interference through shared resources, and missing boundary conditions. This highlights the need to explicate assumptions on the design context in OEM-supplier commercial contracts.

Multiple lines of attack have been developed to cope with the above difficulties.

Regarding holistic approaches, the iterative and incremental development [134] was first proposed several decades ago. More recently, of particular interest to the development of embedded systems were: the V-model process, component-based design and model-based development [165, 130, 107, 47, 163, 35, 164, 172, 17, 190, 79], virtual integration and Platform-Based Design [175, 93, 191, 101, 78].

Another key answer to the complexity of OEM-supplier chains has been standardization. Standardization concerns both the design entities as well as the design processes, particularly through the mechanism of certification. Examples of these standards in the automotive sector include the recently approved requirement interchange format standard RIF[2], the AUTOSAR[3] de-facto standard, the OSEK[4] operating system standard, standardized bus-systems such as CAN[5] and Flexray[6], standards for "car2X" communication, and standardized representations of test supported by ASAM[7]. Examples in the aerospace domain include ARINC standards[8] such as the avionics applications standard interface, IMA, and RTCA[9] communication standards. In the automation domain, standards for interconnection of automation devices such as Profibus[10] are complemented by standardized design languages for application development such as Structured Text. 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. Shared use of Product Lifecycle Management (PLM)[11] 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[12] for the automotive domain, IEC 61508[13] for automation, its derivatives

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

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

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

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

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

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

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

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

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

[11] PLM: Product Lifecycle Management [179]. 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.

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

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

Cenelec EN 50128 and 50126[14] for rail, and Do 178 B[15] for civil avionics.

Requirement capture and engineering is the design activity where relations between actors of the supply chain are legally and technically formalized. It plays thus a key role in managing system integration well. Efforts have been made by paying close attention to book keeping activities, i.e., the management of the requirement descriptions and corresponding traceability support (e.g., using commercial tools such as Doors[16] in combination with Reqtify[17]) and by inserting whenever possible precise formulation and analysis methods and tools.

The way system design challenges have been addressed so far leaves huge opportunities for improvements by relying on *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 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. 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. We now briefly summarize how contracts could improve the current situation in different ways.

Contribution 1 *Addressing the Complexity of Systems.*

While component based design has been a critical step in breaking systems complexity, it does not by itself provide the ultimate answer. When design is being performed at a considered layer, implicit—and generally hidden—assumptions regarding other layers (e.g., computing resources) are typically invoked by the designer. Actual properties of these other layers, however, 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. By pinpointing responsibilities and making hidden assumptions explicit, contract-based design provides the due discipline, concepts, and techniques to cope with this. One challenge for component-based design of embedded systems is to provide interface specifications that address behaviors, not only type properties of interfaces, and are rich enough to cover all phases of the design cycle. This calls for including non-functional characteristics as part of the component interface specifications, which is best achieved by using multiple viewpoints [28, 34]. Contract-based design supports multiple viewpoints by giving a mathematically precise answer to what it means to fuse them.

Contribution 2 *Addressing OEM-Supplier Chains.*

The problems raised by the complexity of OEM-Supplier Chains are indeed the core target of contract-based design. By making the explication of implicit assumptions mandatory, contracts help assign responsibilities to a precise stake holder for each design entity. By supporting independent development of the different sub-systems while guaranteeing smooth system integration, they orthogonalize the development of complex systems. Contracts are thus adequate candidates for a technical counterpart of the

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

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

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

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

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 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.

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.

The organization of the paper is as follows. In Section 2 we first discuss the requirements on a theory of contracts, based on methodological considerations, particularly the need to support different viewpoints on the system (operation, function, timing, energy, safety, etc.) and to allow for independent development by suppliers. Then we develop a primer on contracts by using a very simple example requiring only elementary mathematical background to be followed. The purpose of this simplistic example 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. This section concludes with a general bibliography on contract based design in general. Section 3 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, it 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. The meta-theory by itself is non-effective in that it does not specify how components and contracts are effectively represented and manipulated. The subsequent series of sections propose a panorama of major concrete contract frameworks. Section 4 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 is flexible in that it allows for different styles of description of such properties—computational efficiency depends on the style adopted. Section 5 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.

This paper contains no illustration example. This is developed in the companion paper [30] where an application case for requirement engineering is presented in Section 3 and another one related to real-time scheduling in the context of AUTOSAR is discussed in Section 4.

2 Contracts: what? 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 motivate by methodological considerations the algebra we need on contracts. We then study a simple instance of this algebra on a toy example, where all operations can be examplified. We conclude the section by providing a (non exhaustive) bibliography on the general concept of contract.

2.1 Contract based design

Contract based design can be seen as a set of methodological guidelines exploiting an algebra of contracts characterized by the operators of refinement $\preceq$, conjunction $\wedge$, and composition $\otimes$. In this section we review these guidelines and discuss the requirements they set about the contract algebra.

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

Managing Requirements and Fusing Viewpoints: 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.

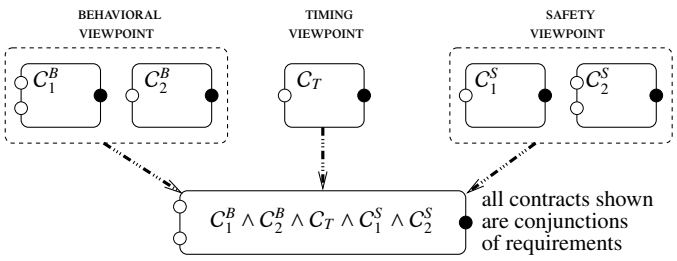


Figure 1: Conjunction of requirements and viewpoints in top-level design

This process is illustrated in Figure 1. 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 get the full system specification, the different viewpoints must be fused. As the notation of Figure 1 suggests, 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. This wrongly suggests that the usual logical conjunction is used. In fact, the need to handle differently guarantees and assumptions will make this notion of "conjunction" subtle.

Design Chain Management, Reuse, and Independent Development: In Figure 2, 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 1. As a first design step, the

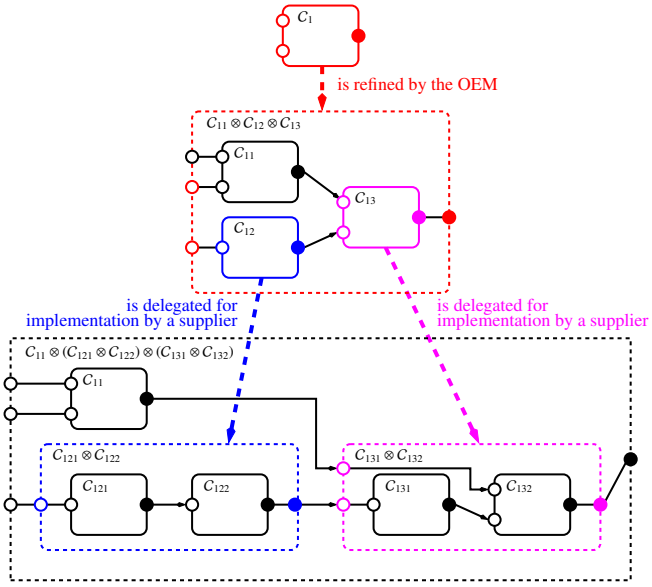


Figure 2: Stepwise refinement

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

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

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

> *if designs are independently performed for each sub-contract $\mathscr{C}_{1j}$, for $j = 1, 2, 3$, then integrating these subsystems yields an implementation that satisfies the composed contract $\mathscr{C}_{11} \otimes \mathscr{C}_{12} \otimes \mathscr{C}_{13}$.* (1)

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

$$\mathscr{C}_{11} \otimes \mathscr{C}_{12} \otimes \mathscr{C}_{13} \quad \preceq \quad \mathscr{C}_1 \tag{2}$$

Overall, the satisfaction of (2) guarantees the correctness of this first design step performed by the OEM.

Obtaining the three sub-contracts $\mathscr{C}_{11}$, $\mathscr{C}_{12}$, and $\mathscr{C}_{13}$, 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 (2) holds for the decomposition step or not.
- In passing, the compatibility of the three sub-contracts $\mathscr{C}_{11}$, $\mathscr{C}_{12}$, and $\mathscr{C}_{13}$, 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 subsystem in a way that subsequent system integration will be correct.

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

Alternatively, this supplier can develop some subsystems by reusing off-the-shelf components. For example, contract $\mathscr{C}_{121}$ 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 $\mathscr{C}_{121}$? In this context, *shared implementations* are of interest. This is illustrated on Figure 3 where the same off-the-

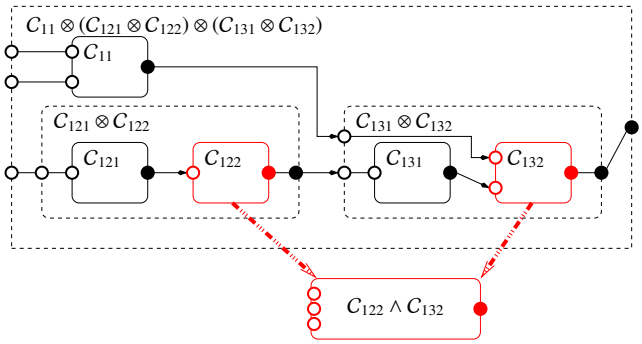


Figure 3: Conjunction for component reuse

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* (1) *ensures that independent development holds.*

2.2 A primer on contracts

In this section we instantiate the above motivated algebra on a very simple framework of "stateless contracts" where the properties considered do not involve system states.

2.2.1 Components, Environments, and 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 unwanted input $y = 0$ by generating an arbitrary output (0 in this case):

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

A *contract*, denoted by the symbol $\mathscr{C}$, is a way of specifying components with the following characteristic properties:

1. Contracts are intentionally abstract;
2. Contracts distinguish responsibilities of a component from those of its environment.

By 1, contracts expose enough information about the component, but not more than necessary for the intended purpose. We can see a contract as an under-specified description of a component that can either be very close to the actual component, or specify only a single property of a component behavior. Regarding 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 $\mathscr{C}$ at its most abstract level as a pair $\mathscr{C} = (\mathcal{E}_{\mathscr{C}}, \mathcal{M}_{\mathscr{C}})$ of subsets of components that implement the contract and of subsets of environments in which the contract can operate. We say that a contract $\mathscr{C}$ is *consistent* if $\mathcal{M}_{\mathscr{C}} \neq \emptyset$ and *compatible* if $\mathcal{E}_{\mathscr{C}} \neq \emptyset$.

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:

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

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

"assumptions $\Rightarrow$ guarantees"

i.e., for the above example, $y \neq 0 \Rightarrow z = x/y$. Intuitively, contract $\mathscr{C}_1$ 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 $\mathcal{E}_{\mathscr{C}_1}$ 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 $\mathscr{C}_1$ should cope with the illegal input $y = 0$. In particular, we have that M_1 implements $\mathscr{C}_1$. Thus, contract $\mathscr{C}_1$ is consistent. We now show a variant of contract $\mathscr{C}_1$ that is not consistent:

$$\mathscr{C}_1' : \begin{cases} \text{variables:} & \begin{cases} \text{inputs:} & x, y \\ \text{outputs:} & z \end{cases} \\ \text{types:} & x, y, z \in \mathbb{R} \\ \text{assumptions:} & \text{T} \\ \text{guarantees:} & z = x/y \end{cases}$$

where symbol T denotes the boolean constant "true". In contrast to $\mathscr{C}_1$, the contract $\mathscr{C}_1'$ makes no assumption on values of the input y. Hence, every component that implements $\mathscr{C}_1'$ has to compute the quotient x/y for all values of y, including $y = 0$, which makes no sense.

2.2.2 Contract Operators

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

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

Consider the component M_2, defined as follows:

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

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

Now, consider component M_2' that consists of an input variable x and an output variable z, both of type real, where $z = abs(x)$ denotes the absolute value of x:

$$M_2' : \begin{cases} \text{variables:} & \begin{cases} \text{inputs:} & x \\ \text{outputs:} & z \end{cases} \\ \text{types:} & x, z \in \mathbb{R} \\ \text{behaviors:} & z = abs(x) \end{cases}$$

Component M_2' is not composable with M_1, because the two components share the same output variable z. Their composition is 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 $\mathscr{C}$ and $\mathscr{C}'$ are *compatible* if their variable types match and if there exists an environment in which the two contracts properly interact. The resulting composition $\mathscr{C} \otimes \mathscr{C}'$ 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 $\mathscr{C}$ a contract, let $A_{\mathscr{C}}$ and $G_{\mathscr{C}}$ be its assumptions and guarantees and define

$$\begin{aligned} G_{\mathscr{C}_1 \otimes \mathscr{C}_2} &= G_{\mathscr{C}_1} \wedge G_{\mathscr{C}_2} \\ A_{\mathscr{C}_1 \otimes \mathscr{C}_2} &= \max \left\{ A \;\middle|\; \begin{array}{l} A \wedge G_{\mathscr{C}_2} \Rightarrow A_{\mathscr{C}_1} \\ \text{and} \\ A \wedge G_{\mathscr{C}_1} \Rightarrow A_{\mathscr{C}_2} \end{array} \right\} \end{aligned} \tag{3}$$

where "max" refers to the order of predicates by implication; thus $A_{\mathscr{C}_1 \otimes \mathscr{C}_2}$ 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 $\mathscr{C}_1$ and $\mathscr{C}_2$ are called *compatible* if the assumption computed as in (3) differs from F, the "false" predicate.

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

$$\mathscr{C}_2 : \begin{cases} \text{variables:} & \begin{cases} \text{inputs:} & u \\ \text{outputs:} & x \end{cases} \\ \text{types:} & u, x \in \mathbb{R} \\ \text{assumptions:} & \top \\ \text{guarantees:} & x > u \end{cases} \quad \text{and} \quad \mathscr{C}_2' : \begin{cases} \text{variables:} & \begin{cases} \text{inputs:} & v \\ \text{outputs:} & y \end{cases} \\ \text{types:} & v, y \in \mathbb{R} \\ \text{assumptions:} & \top \\ \text{guarantees:} & y = -v \end{cases}$$

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

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

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

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

Both compositions possess a non-empty assumption, reflecting that the two pairs $(\mathscr{C}_1, \mathscr{C}_2)$ and $(\mathscr{C}_1, \mathscr{C}_2')$ are compatible.

In our example, it holds that contract composition is associative and commutative, that is, compositions $\mathscr{C}_1 \otimes (\mathscr{C}_2 \otimes \mathscr{C}_3)$ and $(\mathscr{C}_1 \otimes \mathscr{C}_2) \otimes \mathscr{C}_3$ result in equivalent contracts, as well as compositions $\mathscr{C}_1 \otimes \mathscr{C}_2$ and $\mathscr{C}_2 \otimes \mathscr{C}_1$, thus providing support for incremental system integration. This result will follow from the results of Section 3 on the meta-theory.

A *quotient* operation can be defined that is dual to the composition operation. Given a system-wide contract $\mathscr{C}$ and a contract $\mathscr{C}_1$ that specifies pre-existing components and their interactions, the quotient operation $\mathscr{C}/\mathscr{C}_1$ 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.

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 $\mathscr{C}'$ refines contract $\mathscr{C}$, then *any implementation of $\mathscr{C}'$ should 1) implement $\mathscr{C}$ and, 2) be able to operate in any environment for $\mathscr{C}$*. Hence the following definition for the refinement pre-order $\preceq$ between contracts: we say that the contract $\mathscr{C}'$ refines the contract $\mathscr{C}$, if $\mathcal{E}_{\mathscr{C}'} \supseteq \mathcal{E}_{\mathscr{C}}$ and $\mathcal{M}_{\mathscr{C}'} \subseteq \mathcal{M}_{\mathscr{C}}$. Since $\preceq$ is a pre-order, refinement is a transitive relation. For our current series of examples, and using previous notations, $\mathscr{C}' \preceq \mathscr{C}$ amounts to requiring that 1) $A_{\mathscr{C}}$ implies $A_{\mathscr{C}'}$, and 2) $A_{\mathscr{C}'} \Rightarrow G_{\mathscr{C}'}$ implies $A_{\mathscr{C}} \Rightarrow G_{\mathscr{C}}$. Also, for all contracts $\mathscr{C}_1$, $\mathscr{C}_2$, $\mathscr{C}_1'$ and $\mathscr{C}_2'$, if $\mathscr{C}_1$ is compatible with $\mathscr{C}_2$ and $\mathscr{C}_1' \preceq \mathscr{C}_1$ and $\mathscr{C}_2' \preceq \mathscr{C}_2$, then $\mathscr{C}_1'$ is compatible with $\mathscr{C}_2'$ and $\mathscr{C}_1' \otimes \mathscr{C}_2' \preceq \mathscr{C}_1 \otimes \mathscr{C}_2$.

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

$$\mathscr{C}_1'' : \begin{cases} \text{variables:} & \begin{cases} \text{inputs:} & y \\ \text{outputs:} & z \end{cases} \\ \text{types:} & y, z \in \mathbb{R} \\ \text{assumptions:} & y \neq 0 \\ \text{guarantees:} & z \in \mathbb{R} \end{cases} \quad \text{and} \quad \mathscr{C}_2'' : \begin{cases} \text{variables:} & \begin{cases} \text{inputs:} & x \\ \text{outputs:} & y \end{cases} \\ \text{types:} & x, y \in \mathbb{R} \\ \text{assumptions:} & \text{T} \\ \text{guarantees:} & y > 0 \end{cases}$$

The contract $\mathscr{C}_1''$ 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 $\mathscr{C}_1''$ does not declare the nominator input variable x. The contract $\mathscr{C}_2''$ specifies components that have an input variable x and an output variable of type y. The only requirement on the behavior of $\mathscr{C}_2''$ is that y is strictly greater than 0. The composition $\mathscr{C}_1'' \otimes \mathscr{C}_2''$ is well defined. The contract $\mathscr{C}_1$ refines $\mathscr{C}_1''$, 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 $\mathscr{C}_1$ is also compatible with $\mathscr{C}_2''$ and that $\mathscr{C}_1 \otimes \mathscr{C}_2'' \preceq \mathscr{C}_1'' \otimes \mathscr{C}_2''$. Finally, we have that M_1 and M_2 are implementations of their respective contracts. It follows that $M_1 \times M_2$ implements $\mathscr{C}_1 \otimes \mathscr{C}_2''$.

Contract Conjunction and Viewpoint Fusion: We now introduce the *conjunction* operator between contracts, denoted by the symbol $\wedge$. Conjunction complements 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.
3. Conjunction supports reuse of a component in different parts of a design.

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

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

$$\mathscr{C}_1^T : \begin{cases} \text{variables:} & \begin{cases} \text{inputs:} & t_x, t_y \\ \text{outputs:} & t_z \end{cases} \\ \text{types:} & t_x, t_y, t_z \in \mathbb{R}_+ \\ \text{assumptions:} & \text{T} \\ \text{guarantees:} & t_z \leq \max(t_x, t_y) + 1 \end{cases}$$

The contract $\mathscr{C}_1^T$ is consistent with $\mathscr{C}_1$, meaning that $\mathscr{C}_1^T \wedge \mathscr{C}_1$ possesses implementations. Their conjunction $\mathscr{C}_1 \wedge \mathscr{C}_1^T$ yields a contract that guarantees, in addition to $\mathscr{C}_1$ 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 τ assumption of the timing contract, and cancels the assumption of $\mathscr{C}_1$. 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 4. For the moment, we can fix the problem by introducing y in the interface of the contract, and use it in the assumptions, as in the following contract $\mathscr{C}_2^T$

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

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 4 and subsequent ones.

2.3 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". This bibliographical note is limited to the grounding work on contract based design, across the different communities that have considered the problem, namely: software engineering, language design, system engineering, and formal methods in a broad sense. We report here a partial and limited overview of how this paradigm has been tackled in these different communities. While we do not claim being exhaustive, we hope that the reader will find her way to the different literatures. This note is organized into two parts. In the second part we focus on the development of contract based design for embedded systems and Cyber-Physical systems, which is the main focus of this tutorial. In a first part, we review the work done by the other communities, under the generic name of "SW engineering". A more extensive and deeper coverage is given in subsequent bibliographical notes, for the different subtopics discussed in the different sections.

2.3.1 Contracts in SW engineering

This part of the bibliographical note was inspired by the report [188]. Design by Contract is a software engineering technique popularized by Bertrand Meyer [155, 156] following earlier ideas from Floyd-Hoare logic [189, 125]. 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 satisfies 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. [40]. 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 [158]. 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 [126] 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) [131, 193, 146]. 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) [166, 202]. Roughly speaking, an OCL statement refers to a context for the considered statement, and expresses properties to be satisfied by this context (e.g., if the context is a class, a property might be an attribute). Arithmetic or set-theoretic operations can be used in expressing these properties. OCL can, for instance, be used to specify an invariant in terms of the conditions that must be satisfied *before* and *after* the execution of a method or the step of a state machine, providing ways to express assumptions and guarantees. Likewise, attributes on port methods have been used to represent non-functional requirements or provisions of a component [58, 57]. 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 [18, 147, 178]. The notion of refinement is replaced by the concept of class inheritance. From a contract theory point of view, this approach has several limitations. Inheritance, for instance, does not properly cover aspects related to behavior refinement, in the sense that the abstract class is unable to constrain the actions that its implementations may perform, and is instead limited to establishing the signature of the methods. Nor is it made precise what it means to take the conjunction of interfaces, which can only be approximated by multiple inheritance, or to compose them.

In a continuing effort since his joint work with W. Damm on *Live Sequence Charts* (LSC) in 2000 [77] with its *Play-Engine* implementation [120], David Harel has developed the concept of *behavioral programming* [121, 119, 122], 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.

2.3.2 Our focus—contracts for systems and CPS

The frameworks of contracts developed in the area of Software Engineering were 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* [118, 123, 114, 152] are encountered, that is systems that continuously interact with some environment, as opposed to *transformational systems* [114], 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 [165] or AADL [168] for system level modeling, Modelica [107] for physical system modeling, Matlab-Simulink [130] for control-law design, and Scade [161, 35] and TargetLink for detailed software design. UML-related standardization efforts in this area also include the MARTE UML[18] 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. Abadi and Lamport (with Wolper for the first publication) [3, 1] were the first to propose a comprehensive Assume/Guarantee specification theory for Transition Systems. The first publication introduced the game point of view in distinguishing component from environment and using this for defining refinement. The second long publication proposed a comprehensive specification framework with Assumptions (restricted to safety properties) and Guarantees (both safety and liveness are covered). The composition of specifications is studied and the issue of circular reasoning is pinpointed and solved by a reinforcement of the assumptions of the composition, with comparison to the "intuitive" definition for them. The resulting theory is quite complex, which may explain why it did not deserve the attention it should. Dill proposed *asynchronous trace structures* with failure behaviors [94]. 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.

[18] www.omgmarte.org

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 [203]. Negulescu later generalizes the algebra to Process Spaces which abstract away the specifics of the behaviors, and derives new composition operators [160]. This particular abstraction technique was earlier introduced by Burch with Trace Algebras to construct conservative approximations [53], and later generalized by Passerone and Burch [54, 171] to study generic trace structures with failure behaviors and to formalize the problem of computing the quotient (there called mirror) [169]. Methodological aspects of contract-based design of Cyber-Physical Systems are extensively discussed in [192]. 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.

3 A Mathematical Meta-Theory of Contracts

In Section 2 and its bibliographical discussion, we showed that a number of frameworks—specification, interface, contract theories, and more—were proposed to cope with the issues of system development in a supplier chain. The list of such frameworks will be further increased in the forthcoming sections. This calls for developing a "birds-eye view" of the subject, by which the essence and commonalities of such frameworks will be highlighted. In software engineering, meta-models are "models of models", i.e., formal ways of specifying a certain family of models [190, 172]. Analogously, we call *meta-theory* a way to specify a particular family of theories. In this section we propose a meta-theory of contracts.

Our meta-theory is summarized in Table 2. 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.

In this meta-theory we will mainly focus on semantic concepts. Clearly, components, contracts, and the associated relations and operators, must be expressed in some language which defines the syntax for specifying them. The properties of the specification language are important in several respects. In particular, the finite nature of the representation may limit the kind of objects that can be represented, and could consequently affect the realizability (closure) of the operators of our theory—we will pay special attention to this issue. Nonetheless, in this section on the "meta-theory", we are interested primarily in the relations between the objects that the language describes, irrespective of how they are represented. Therefore, we will proceed under the hypothesis that questions of representations have been adequately addressed. Hence, when referring to components and contracts, we implicitly assume that they are described in some language whose semantics maps to the concepts that we present in this section. How this meta-theory can be instantiated to various concrete frameworks is discussed in subsequent sections.

The meta-theory we develop here is novel. There are very few attempts of that kind. In fact, the only ones we are aware of are the recent works by Bauer et al. [19] and Chen et al. [66], which follow a different (and complementary) approach. The discussion of this work is deferred to the bibliographical Section 3.8.

Concept	**Definition and generic properties**
Primitive	
Component	Components are denoted by M
Composability of components	A type property on pairs of components (M_1, M_2)
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
Environment	An *environment* for component M is a component E such that $E \times M$ is well defined
Derived	
Contract	The semantics of *contract* $\mathscr{C}$ is a pair $(\mathcal{E}_{\mathscr{C}}, \mathcal{M}_{\mathscr{C}})$, where $\mathcal{M}_{\mathscr{C}}$ is a subset of components and $\mathcal{E}_{\mathscr{C}}$ a subset of valid environments
Consistency	$\mathscr{C}$ is *consistent* iff it has at least one component: $\mathcal{M}_{\mathscr{C}} \neq \emptyset$
Compatibility	$\mathscr{C}$ is *compatible* iff it has at least one environment: $\mathcal{E}_{\mathscr{C}} \neq \emptyset$
Implementation	$M \models^{\mathrm{M}} \mathscr{C}$ if and only if $M \in \mathcal{M}_{\mathscr{C}}$ $E \models^{\mathrm{E}} \mathscr{C}$ if and only if $E \in \mathcal{E}_{\mathscr{C}}$
Refinement	$\mathscr{C}' \preceq \mathscr{C}$ iff $\mathcal{E}_{\mathscr{C}'} \supseteq \mathcal{E}_{\mathscr{C}}$ and $\mathcal{M}_{\mathscr{C}'} \subseteq \mathcal{M}_{\mathscr{C}}$; Property 1 holds
GLB and LUB of contracts	$\mathscr{C}_1 \wedge \mathscr{C}_2$ = Greatest Lower Bound (GLB) of $\mathscr{C}_1$ and $\mathscr{C}_2$ for $\preceq$; $\mathscr{C}_1 \vee \mathscr{C}_2$ = Least Upper Bound (LUB) of $\mathscr{C}_1$ and $\mathscr{C}_2$ for $\preceq$; Assumption 1 is in force and Property 2 holds Say that $(\mathscr{C}_1, \mathscr{C}_2)$ is a *consistent pair* if $\mathscr{C}_1 \wedge \mathscr{C}_2$ is consistent
Composition of contracts	$\mathscr{C}_1 \otimes \mathscr{C}_2$ is well defined if $\left.\begin{array}{r}\forall M_1 \models^{\mathrm{M}} \mathscr{C}_1 \\ \forall M_2 \models^{\mathrm{M}} \mathscr{C}_2\end{array}\right\} \Rightarrow (M_1, M_2)$ composable $\mathscr{C}_1 \otimes \mathscr{C}_2 = \min\left\{ \mathscr{C} \;\middle|\; \left[\begin{array}{r}\forall M_1 \models^{\mathrm{M}} \mathscr{C}_1 \\ \text{and } \forall M_2 \models^{\mathrm{M}} \mathscr{C}_2 \\ \text{and } \forall E \models^{\mathrm{E}} \mathscr{C}\end{array}\right] \Rightarrow \left[\begin{array}{c}M_1 \times M_2 \models^{\mathrm{M}} \mathscr{C} \\ \text{and } E \times M_2 \models^{\mathrm{E}} \mathscr{C}_1 \\ \text{and } E \times M_1 \models^{\mathrm{E}} \mathscr{C}_2\end{array}\right] \right\}$ Assumption 2 is in force; Properties 3, 4, 5, and Corollaries 1 and 2 hold Say that $(\mathscr{C}_1, \mathscr{C}_2)$ is a *compatible pair* if $\mathscr{C}_1 \otimes \mathscr{C}_2$ is compatible
Quotient	$\mathscr{C}_1 / \mathscr{C}_2 = \bigvee\{\mathscr{C} \mid \mathscr{C} \otimes \mathscr{C}_2 \preceq \mathscr{C}_1\}$; Property 6 holds

Table 2: Summary of the meta-theory of contracts. We first list **primitive** concepts and then **derived** concepts introduced by the meta-theory. $\mathscr{C}_1$, $\mathscr{C}_2$, and $\mathscr{C}$ are implicitly universally quantified over some underlying class **C** of contracts which depends on the particular contract framework considered.

3.1 Components and their composition

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

We assume a *composition* operator $M_1 \times M_2$ acting on pairs of components. Component composition $\times$ is partially, not totally, defined. Two components such that $M_1 \times M_2$ is well defined are called *composable*. Composability of components is meant to be a typing property. 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. An *environment* for a component M is another component E composable with M.

3.2 Contracts

In our primer of Section 2, we have highlighted the importance of the valid environments associated with contracts, for which an implementation will operate satisfactorily. At the abstract level of the meta-theory, we make this explicit next:

Definition 1 *We consider a class* **C** *of* contracts $\mathscr{C}$ *whose semantics is a pair* $[\![\mathscr{C}]\!] = (\mathcal{E}_{\mathscr{C}}, \mathcal{M}_{\mathscr{C}}) \in 2^{\mathbb{M}} \times 2^{\mathbb{M}}$*, where:*

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

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

$$M \models^{\mathrm{M}} \mathscr{C} \quad \text{and} \quad E \models^{\mathrm{E}} \mathscr{C}$$

to express that $M \in \mathcal{M}_{\mathscr{C}}$ *and* $E \in \mathcal{E}_{\mathscr{C}}$*, respectively.*

In the meta-theory the class **C** is abstract. Each particular contract framework comes with a concrete definition of **C** and instantiates all the concepts listed in the last column of Table 2, thus making them effective. While the definition of contract consistency complies with the intuition, the definition of contract compatibility may seem strange at first glance. We shall, however, see later that it specializes to known notions for concrete theories.

3.3 Refinement and conjunction

To support independent implementability, the concept of contract refinement must ensure the following: if contract $\mathscr{C}'$ refines contract $\mathscr{C}$, then any implementation of $\mathscr{C}'$ should implement $\mathscr{C}$ and be able to operate in any environment for $\mathscr{C}$. Hence the following definition for the refinement preorder $\preceq$ between contracts: $\mathscr{C}'$ refines $\mathscr{C}$,

written $\mathscr{C}' \preceq \mathscr{C}$, if and only if $\mathcal{M}_{\mathscr{C}'} \subseteq \mathcal{M}_{\mathscr{C}}$ and $\mathcal{E}_{\mathscr{C}'} \supseteq \mathcal{E}_{\mathscr{C}}$. 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 $\mathscr{C}'$ is an implementation of $\mathscr{C}$: $M \models^{\mathrm{M}} \mathscr{C}' \Rightarrow M \models^{\mathrm{M}} \mathscr{C}$;*
2. *Any environment of $\mathscr{C}$ is an environment of $\mathscr{C}'$: $E \models^{\mathrm{E}} \mathscr{C} \Rightarrow E \models^{\mathrm{E}} \mathscr{C}'$.*

At this point we need the following assumption on the contract language:

Assumption 1 *For $\mathbf{C}' \subseteq \mathbf{C}$ any subset of expressible contracts, the Greatest Lower Bound (GLB) $\bigwedge \mathbf{C}'$ and the Least Upper Bound (LUB) $\bigvee \mathbf{C}'$ both exist in $\mathbf{C}$, where GLB and LUB refer to refinement order.*

Although strong, this assumption is satisfied by the instances of contract languages we know, see the subsequent sections for this. It allows us to define the *conjunction* $\mathscr{C}_1 \wedge \mathscr{C}_2$ of contracts $\mathscr{C}_1$ and $\mathscr{C}_2$ as the GLB of these two contracts. The intent is to define this conjunction as the intersection of sets of implementations and the union of sets of environments. However, not every pair of sets of components can be the semantics of a contract belonging to class $\mathbf{C}$. 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 $\mathscr{C}_1 \wedge \mathscr{C}_2$ also refines $\mathscr{C}_1$ and $\mathscr{C}_2$. Any implementation of $\mathscr{C}_1 \wedge \mathscr{C}_2$ is a shared implementation of $\mathscr{C}_1$ and $\mathscr{C}_2$. Any environment of $\mathscr{C}_1$ or $\mathscr{C}_2$ is an environment of $\mathscr{C}_1 \wedge \mathscr{C}_2$.*
2. *For $\mathbf{C}' \subseteq \mathbf{C}$ a subset of contracts, $\bigwedge \mathbf{C}'$ is compatible if and only if there exists a compatible $\mathscr{C} \in \mathbf{C}'$.*

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

3.4 Contract composition

On top of component composition, we define a *contract composition* $\mathscr{C}_1 \otimes \mathscr{C}_2$, whose intuition is as follows: composing two implementations of $\mathscr{C}_1$ and $\mathscr{C}_2$ should yield an implementation of $\mathscr{C}_1 \otimes \mathscr{C}_2$ and any environment for $\mathscr{C}_1 \otimes \mathscr{C}_2$, when composed with an implementation for $\mathscr{C}_1$, should yield a valid environment for $\mathscr{C}_2$ and vice-versa. Observe that $E \models^{\mathrm{E}} \mathscr{C}$ implies that E is composable with any implementation of $\mathscr{C}$, and thus $E \times M_i$ are well defined. Formally, $\mathscr{C}_1 \otimes \mathscr{C}_2$ is defined by the formula given in Table 2, where "min" refers to the refinement order. For this to make sense, we assume the following:

Assumption 2 *We assume that the* min *in the formula defining $\mathscr{C}_1 \otimes \mathscr{C}_2$ in Table 2 exists and is unique.*

Denote by $\mathbf{C}_{\mathscr{C}_1,\mathscr{C}_2}$ the set of contracts defined by the brackets in the formula defining $\mathscr{C}_1 \otimes \mathscr{C}_2$ in Table 2, that is:

$$\mathbf{C}_{\mathscr{C}_1,\mathscr{C}_2} \quad =_{\text{def}} \quad \left\{ \mathscr{C} \;\middle|\; \left[\begin{array}{r} \forall M_1 \models^{\text{M}} \mathscr{C}_1 \\ \text{and } \forall M_2 \models^{\text{M}} \mathscr{C}_2 \\ \text{and } \forall E \;\; \models^{\text{E}} \mathscr{C} \end{array} \right] \Longrightarrow \left[\begin{array}{c} M_1 \times M_2 \models^{\text{M}} \mathscr{C} \\ \text{and } E \times M_2 \models^{\text{E}} \mathscr{C}_1 \\ \text{and } E \times M_1 \models^{\text{E}} \mathscr{C}_2 \end{array} \right] \right\} \tag{4}$$

With this notation, Assumption 2 rewrites $\bigwedge \mathbf{C}_{\mathscr{C}_1,\mathscr{C}_2} \in \mathbf{C}_{\mathscr{C}_1,\mathscr{C}_2}$ and contract composition rewrites

$$\mathscr{C}_1 \otimes \mathscr{C}_2 \quad = \quad \bigwedge \mathbf{C}_{\mathscr{C}_1,\mathscr{C}_2} \tag{5}$$

The following lemma will be instrumental:

Lemma 1 *Let four contracts be such that $\mathscr{C}'_1 \preceq \mathscr{C}_1$, $\mathscr{C}'_2 \preceq \mathscr{C}_2$, and $\mathscr{C}_1 \otimes \mathscr{C}_2$ is well defined. Then, so is $\mathscr{C}'_1 \otimes \mathscr{C}'_2$ and*

$$\mathbf{C}_{\mathscr{C}'_1,\mathscr{C}'_2} \supseteq \mathbf{C}_{\mathscr{C}_1,\mathscr{C}_2}$$

Proof: Since $\mathscr{C}_1 \otimes \mathscr{C}_2$ 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, $\mathscr{C}'_1 \otimes \mathscr{C}'_2$ is well defined according to the formula of Table 2. Next, since $\mathscr{C}'_1 \preceq \mathscr{C}_1$ and $\mathscr{C}'_2 \preceq \mathscr{C}_2$, $M_1 \models^{\text{M}} \mathscr{C}'_1$ and $M_2 \models^{\text{M}} \mathscr{C}'_2$ implies $M_1 \models^{\text{M}} \mathscr{C}_1$ and $M_2 \models^{\text{M}} \mathscr{C}_2$; similarly $E \times M_2 \models^{\text{E}} \mathscr{C}_1$ and $E \times M_1 \models^{\text{E}} \mathscr{C}_2$ implies $E \times M_2 \models^{\text{E}} \mathscr{C}'_1$ and $E \times M_1 \models^{\text{E}} \mathscr{C}'_2$. Therefore, replacing, in the right hand side of (4), $\mathscr{C}_1$ by $\mathscr{C}'_1$ and $\mathscr{C}_2$ by $\mathscr{C}'_2$ can only increase the set $\mathbf{C}_{\mathscr{C}_1,\mathscr{C}_2}$. □

To conform to the usage, we say that $\mathscr{C}_1$ and $\mathscr{C}_2$ are *compatible* contracts if their composition $\mathscr{C}_1 \otimes \mathscr{C}_2$ is defined and compatible in the sense of Table 2. The following properties are a direct corollary of Lemma 1:

Property 3 (independent implementability) *For all contracts $\mathscr{C}_1$, $\mathscr{C}_2$, $\mathscr{C}'_1$ and $\mathscr{C}'_2$, if 1) $\mathscr{C}_1$ is compatible with $\mathscr{C}_2$, and 2) $\mathscr{C}'_1 \preceq \mathscr{C}_1$ and $\mathscr{C}'_2 \preceq \mathscr{C}_2$ hold, then $\mathscr{C}'_1$ is compatible with $\mathscr{C}'_2$ and $\mathscr{C}'_1 \otimes \mathscr{C}'_2 \preceq \mathscr{C}_1 \otimes \mathscr{C}_2$.*

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

Corollary 1 *Compatible contracts can be independently implemented.*

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

This is in contrast with *compositional verification*, where $\times_{i\in I} M_i \models^{\text{M}} P$ is to be checked, where M_i are detailed implementations and P is a property. In this context, I may be a large set, and thus 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 [196, 71, 112, 2, 132].

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

Property 4 *For all contracts $\mathscr{C}_1$, $\mathscr{C}_2$, $\mathscr{C}_3$ and $\mathscr{C}_4$,*

$$(\mathscr{C}_1 \otimes \mathscr{C}_2) \otimes (\mathscr{C}_3 \otimes \mathscr{C}_4) \quad = \quad (\mathscr{C}_3 \otimes \mathscr{C}_1) \otimes (\mathscr{C}_4 \otimes \mathscr{C}_2) \tag{6}$$

Furthermore, if $\mathscr{C}_1$ and $\mathscr{C}_2$ are compatible, $\mathscr{C}_3$ and $\mathscr{C}_4$ are compatible and $\mathscr{C}_1 \otimes \mathscr{C}_2$ is compatible with $\mathscr{C}_3 \otimes \mathscr{C}_4$, then $\mathscr{C}_1$ is compatible with $\mathscr{C}_3$, $\mathscr{C}_2$ is compatible with $\mathscr{C}_4$, $\mathscr{C}_3 \otimes \mathscr{C}_1$ is compatible with $\mathscr{C}_4 \otimes \mathscr{C}_2$.

Proof: To shorten notations, write $\mathscr{C}_{(12)}$ instead of $(\mathscr{C}_1 \otimes \mathscr{C}_2)$, $\mathscr{C}_{(12)(34)}$ instead of the left hand side of (6), and similarly for the other cases. By Assumption 2 and the associativity and commutativity of component composition, $\mathscr{C}_{(12)(34)}$ is characterized by the following two properties, where index i ranges over the set $1\ldots4$:

$$\begin{array}{lcl} M_i \models^{\mathrm{M}} \mathscr{C}_i & \Rightarrow & M_1 \times \ldots \times M_4 \models^{\mathrm{M}} \mathscr{C}_{(12)(34)} \\ E \models^{\mathrm{E}} \mathscr{C}_{(12)(34)} & \Rightarrow & E \times \left(\times_{j \neq i} M_j \right) \models^{\mathrm{E}} \mathscr{C}_i \end{array} \tag{7}$$

Observe that (7) is fully symmetric, which proves (6). For the additional statement, 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 this additional statement. □

By a variation of the same proof, successively, for two contracts $\mathscr{C}_1$ and $\mathscr{C}_3$, and then for three contracts $\mathscr{C}_1$, $\mathscr{C}_3$, and $\mathscr{C}_4$, we get:

Corollary 2 (commutativity, associativity)

- commutativity: $\mathscr{C}_1 \otimes \mathscr{C}_3 = \mathscr{C}_3 \otimes \mathscr{C}_1$;
- associativity: $\mathscr{C}_1 \otimes (\mathscr{C}_3 \otimes \mathscr{C}_4) = (\mathscr{C}_1 \otimes \mathscr{C}_3) \otimes \mathscr{C}_4$.

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

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

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

The use of sub-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, labeled by another index with values 1 or 2 in such a way that contract $\mathscr{C}_{11} \wedge \mathscr{C}_{21}$ is the contract associated with sub-system 1 and $\mathscr{C}_{12} \wedge \mathscr{C}_{22}$ is the contract associated with 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 $\mathscr{C}_{11} \otimes \mathscr{C}_{12}$ and $\mathscr{C}_{21} \otimes \mathscr{C}_{22}$, and, then, taking their conjunction. This property supports independent implementation for specifications involving multiple viewpoints. Observe that only refinement, not equality, holds in (8).

3.5 Quotient

The quotient of two contracts is defined in Table 2. It is the adjoint of the product operation $\otimes$ in that $\mathscr{C}_1/\mathscr{C}_2$ is the most general context $\mathscr{C}$ in which $\mathscr{C}_2$ refines $\mathscr{C}_1$. It formalizes the practice of "patching" a component to make it behave according to another specification. From its definition in Table 2, we deduce the following property:

Property 6 (quotient) *The following holds:*

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

Proof: Immediate, from the definition. □

Discussion. By inspecting Table 2, 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 the derived notions. *Observers* and then *abstractions* we develop next provide generic approaches to recover effectiveness.

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

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

The functions $M \mapsto b^{\mathsf{E}}_{\mathscr{C}}(M)$ and $M \mapsto b^{\mathsf{M}}_{\mathscr{C}}(M)$ being both non-deterministic account 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. The following immediate results hold, regarding consistency and compatibility:

Lemma 2

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

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. To formulate it, we need to compare verdicts. Since verdicts are non-deterministic functions, this must be done with some care: equip the boolean domain with the order $\text{F} < \text{T}$ and say that $b_1^{\text{E}} \leq b_2^{\text{E}}$ if, for any $M \in \mathbb{M}$, at least one of the following two conditions holds:

Case 1**:** there exists a non-decreasing function $\varphi : \{\text{F}, \text{T}\} \to \{\text{F}, \text{T}\}$ such that $b_2^{\text{E}}(M) = \varphi(b_1^{\text{E}}(M))$,

Case 2**:** there exists a non-increasing function $\psi : \{\text{F}, \text{T}\} \to \{\text{F}, \text{T}\}$ such that $b_1^{\text{E}}(M) = \psi(b_2^{\text{E}}(M))$.

The same definition holds for $b_1^{\text{M}} \leq b_2^{\text{M}}$.

Lemma 3 *Let $(b_{\mathscr{C}}^{\text{E}}, b_{\mathscr{C}}^{\text{M}})$ be an observer for $\mathscr{C}$ and let $\mathscr{C}' \preceq \mathscr{C}$. Then, any pair $(b^{\text{E}}, b^{\text{M}})$ satisfying $b^{\text{E}} \geq b_{\mathscr{C}}^{\text{E}}$ and $b^{\text{M}} \leq b_{\mathscr{C}}^{\text{M}}$ is an observer for $\mathscr{C}'$.*

Based on this lemma, Table 3 indicates how operations from the contract algebra can be mirrored to operations on observers.

Notion	**Observer**
$\mathscr{C} = (\mathcal{E}_{\mathscr{C}}, \mathcal{M}_{\mathscr{C}})$	$(b_{\mathscr{C}}^{\text{E}}, b_{\mathscr{C}}^{\text{M}})$
$\mathscr{C} = \mathscr{C}_1 \wedge \mathscr{C}_2$	$b_{\mathscr{C}}^{\text{E}} = b_{\mathscr{C}_1}^{\text{E}} \vee b_{\mathscr{C}_2}^{\text{E}}$, $b_{\mathscr{C}}^{\text{M}} = b_{\mathscr{C}_1}^{\text{M}} \wedge b_{\mathscr{C}_2}^{\text{M}}$
$\mathscr{C} = \mathscr{C}_1 \vee \mathscr{C}_2$	$b_{\mathscr{C}}^{\text{E}} = b_{\mathscr{C}_1}^{\text{E}} \wedge b_{\mathscr{C}_2}^{\text{E}}$, $b_{\mathscr{C}}^{\text{M}} = b_{\mathscr{C}_1}^{\text{M}} \vee b_{\mathscr{C}_2}^{\text{M}}$
$\mathscr{C} = \mathscr{C}_1 \otimes \mathscr{C}_2$	$b_{\mathscr{C}}^{\text{E}}(E) = \bigwedge_{\substack{M_1 \models^{\text{M}} \mathscr{C}_1 \\ M_2 \models^{\text{M}} \mathscr{C}_2}} \left[b_{\mathscr{C}_2}^{\text{E}}(E \times M_1) \wedge b_{\mathscr{C}_1}^{\text{E}}(E \times M_2) \right]$ $b_{\mathscr{C}}^{\text{M}}(M_1 \times M_2) = b_{\mathscr{C}_1}^{\text{M}}(M_1) \wedge b_{\mathscr{C}_2}^{\text{M}}(M_2)$

Table 3: Mirroring the algebra of contracts with observers.

The formula for $b_{\mathscr{C}_1 \otimes \mathscr{C}_2}^{\text{E}}(E)$ requires considering the set of all implementations M_i of $\mathscr{C}_i$. This set is not within the scope of observers (which are only semi-decision procedures). Worse, it cannot be underapproximated by using observers. Underapproximating this set requires using abstractions introduced in the next section, not 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 2 will ensure effectiveness. In subsequent section where concrete theories are reviewed, we indicate, for each theory, how observers can be constructed and how Lemma 2 specializes.

To conclude, observers provide semi-decision procedures to *disprove* properties such as the validity of a component or an environment, or consistency or compatibility. Observers can be complemented by *abstractions* to *prove* the same properties.

3.7 Abstractions

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

In this section we explain how to lift, to contracts, abstraction procedures available on components. In doing so, our objectives are the following:

1. Abstraction for contracts should allow proving refinement, consistency, or compatibility, for any contract or sets of contracts, based on their abstractions;
2. Properties of contracts should be deducible from their abstractions, compositionally with respect to both conjunction and parallel composition;
3. The mechanism of lifting abstractions, from components to contracts should be generic and instantiable for any concrete contract framework.

A large part of this agenda is achieved. Our starting point is a framework for abstracting components. This framework must be rich enough to support abstraction and its opposite operation in a coherent way. A known formalization of this is the notion of Galois connection, which is key in the theory of Abstract Interpretation [72, 73, 74, 157].

A *Galois connection* consists of two *concrete* and *abstract* partially ordered sets $(\mathcal{X}_{\mathbf{c}}, \sqsubseteq_{\mathbf{c}})$ and $(\mathcal{X}_{\mathbf{a}}, \sqsubseteq_{\mathbf{a}})$, and two total monotonic maps:[19]

$$\begin{array}{lcl} \alpha : \mathcal{X}_{\mathbf{c}} \mapsto \mathcal{X}_{\mathbf{a}} & : & \text{the } \textit{abstraction} \\ \gamma : \mathcal{X}_{\mathbf{a}} \mapsto \mathcal{X}_{\mathbf{c}} & : & \text{the } \textit{concretization} \end{array}$$

such that, for any two $X_{\mathbf{c}} \in \mathcal{X}_{\mathbf{c}}$ and $X_{\mathbf{a}} \in \mathcal{X}_{\mathbf{a}}$,

$$X_{\mathbf{c}} \sqsubseteq_{\mathbf{c}} \gamma(X_{\mathbf{a}}) \quad \text{if and only if} \quad \alpha(X_{\mathbf{c}}) \sqsubseteq_{\mathbf{a}} X_{\mathbf{a}} \tag{10}$$

Property (10) is equivalent to any of the following properties:

$$X_{\mathbf{c}} \sqsubseteq_{\mathbf{c}} \gamma \circ \alpha(X_{\mathbf{c}}) \quad ; \quad \alpha \circ \gamma(X_{\mathbf{a}}) \sqsubseteq_{\mathbf{a}} X_{\mathbf{a}} \tag{11}$$

where $\gamma \circ \alpha$ is the composition of the two referred maps: $\gamma \circ \alpha(X_{\mathbf{c}}) =_{\text{def}} \gamma(\alpha(X_{\mathbf{c}}))$. The intent is that $\mathcal{X}_{\mathbf{c}}$ is the concrete domain of interest and $\mathcal{X}_{\mathbf{a}}$ is a simpler and coarser representation of the former, where concrete entities can be approximated. The two orders $\sqsubseteq_{\mathbf{c/a}}$ are interpreted as "is more precise"—for example, if components are specified as sets of behaviors, the preciseness order is simply set inclusion. The Galois connection property (10) relates the preciseness orders in concrete and abstract domains.

Having the above notions at hand, our next step consists in systematically lifting a given Galois connection (α, γ) on components to an abstraction on contracts, as defined in Table 2. Since contracts are defined as pairs consisting of a set of valid environments and a set of valid components, our first task is to lift Galois connections on sets, to abstractions on powersets. Our construction will be using the notion of inverse map, which we recall next. For X and Y two sets, $f : X \rightarrow Y$ a partial function, and $Z \subseteq Y$, define

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

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

The following holds:

$$\begin{array}{rcl} f^{-1}(Z_1 \cap Z_2) & = & f^{-1}(Z_1) \cap f^{-1}(Z_2) \\ f^{-1}(Z_1 \cup Z_2) & = & f^{-1}(Z_1) \cup f^{-1}(Z_2) \end{array} \tag{12}$$

Referring to the previously introduced notations, we consider the sets $\mathcal{X}_{\mathbf{c}}^{<} \subseteq 2^{\mathcal{X}_{\mathbf{c}}}$ and $\mathcal{X}_{\mathbf{a}}^{<} \subseteq 2^{\mathcal{X}_{\mathbf{a}}}$ collecting all ideals[20] of $(\mathcal{X}_{\mathbf{c}}, \sqsubseteq_{\mathbf{c}})$ and $(\mathcal{X}_{\mathbf{a}}, \sqsubseteq_{\mathbf{a}})$, respectively. Equip $\mathcal{X}_{\mathbf{c}}^{<}$ and $\mathcal{X}_{\mathbf{a}}^{<}$ with their inclusion orders $\subseteq_{\mathbf{c}}$ and $\subseteq_{\mathbf{a}}$. The *canonical abstraction*

$$\widehat{\alpha}: \quad (\mathcal{X}_{\mathbf{c}}^{<}, \subseteq_{\mathbf{c}}) \to (\mathcal{X}_{\mathbf{a}}^{<}, \subseteq_{\mathbf{a}})$$

associated to Galois connection (α, γ) is defined by

$$\widehat{\alpha}(\chi_{\mathbf{c}}) \ =_{\text{def}} \ \gamma^{-1}(\chi_{\mathbf{c}}) \tag{13}$$

where $\chi_{\mathbf{c}}$ ranges over $\mathcal{X}_{\mathbf{c}}^{<}$. Definition (13) is sound since γ is monotonic. The following property follows by construction:

$$\forall \chi_{\mathbf{c}} \in \mathcal{X}_{\mathbf{c}}^{<}: \quad \chi_{\mathbf{c}} = \emptyset \implies \widehat{\alpha}(\chi_{\mathbf{c}}) = \emptyset \tag{14}$$

We now instantiate the generic construction (13) by substituting $\mathcal{X}_{\mathbf{c}} \leftarrow \mathbb{M}_{\mathbf{c}}$ and $\mathcal{X}_{\mathbf{a}} \leftarrow \mathbb{M}_{\mathbf{a}}$, where $\mathbb{M}_{\mathbf{c}}$ and $\mathbb{M}_{\mathbf{a}}$ are concrete and abstract domains of components. We assume the following, which expresses that the preciseness orders fit our contract framework:

Assumption 3 *For any concrete contract $\mathscr{C}_{\mathbf{c}} \in \mathbf{C}_{\mathbf{c}}$ with semantics $[\![\mathscr{C}_{\mathbf{c}}]\!] = \langle \mathcal{E}_{\mathscr{C}_{\mathbf{c}}}, \mathcal{M}_{\mathscr{C}_{\mathbf{c}}} \rangle$, both $\mathcal{E}_{\mathscr{C}_{\mathbf{c}}}$ and $\mathcal{M}_{\mathscr{C}_{\mathbf{c}}}$ are downward closed under $\sqsubseteq_{\mathbf{c}}$. The same holds for abstract contracts.*

Assumption 3 is indeed very natural for known contract frameworks, see Section 4.6 and [33] for more details.

By (13) we inherit an abstraction $\widehat{\alpha}$ from $(\mathbb{M}_{\mathbf{c}}^{<}, \subseteq)$ to $(\mathbb{M}_{\mathbf{a}}^{<}, \subseteq)$. Since the semantics of a concrete generic contract $\mathscr{C}_{\mathbf{c}}$ is $[\![\mathscr{C}_{\mathbf{c}}]\!] = \langle \mathcal{E}_{\mathscr{C}_{\mathbf{c}}}, \mathcal{M}_{\mathscr{C}_{\mathbf{c}}} \rangle \in \mathbb{M}_{\mathbf{c}}^{<} \times \mathbb{M}_{\mathbf{c}}^{<}$, we can define the *abstraction* $\overline{\alpha}(\mathscr{C}_{\mathbf{c}})$ of $\mathscr{C}_{\mathbf{c}}$, whose semantics is:

$$[\![\overline{\alpha}(\mathscr{C}_{\mathbf{c}})]\!] \quad =_{\text{def}} \quad \langle \widehat{\alpha}\left(\mathcal{E}_{\mathscr{C}_{\mathbf{c}}}\right), \widehat{\alpha}\left(\mathcal{M}_{\mathscr{C}_{\mathbf{c}}}\right) \rangle \ \in \ \mathbb{M}_{\mathbf{a}}^{<} \times \mathbb{M}_{\mathbf{a}}^{<} \tag{15}$$

$\overline{\alpha}$ defined by (15) is the *canonical abstraction* on contracts associated to the Galois connection (α, γ) on components. The following theorem achieves our first objectives regarding contract abstraction:

Theorem 1 *Let $M_{\mathbf{c}}, E_{\mathbf{c}}, \mathscr{C}_{\mathbf{c}}$ be a concrete component, environment, and contract.*

1. *If $\alpha(M_{\mathbf{c}}) \models_{\mathbf{a}}^{\mathrm{M}} \overline{\alpha}(\mathscr{C}_{\mathbf{c}})$ holds, then $M_{\mathbf{c}} \models_{\mathbf{c}}^{\mathrm{M}} \mathscr{C}_{\mathbf{c}}$ follows.*
   If $\alpha(E_{\mathbf{c}}) \models_{\mathbf{a}}^{\mathrm{E}} \overline{\alpha}(\mathscr{C}_{\mathbf{c}})$ holds, then $E_{\mathbf{c}} \models_{\mathbf{c}}^{\mathrm{E}} \mathscr{C}_{\mathbf{c}}$ follows.
2. *If $\mathscr{C}'_{\mathbf{c}} \preceq_{\mathbf{c}} \mathscr{C}_{\mathbf{c}}$ holds, then $\overline{\alpha}(\mathscr{C}'_{\mathbf{c}}) \preceq_{\mathbf{a}} \overline{\alpha}(\mathscr{C}_{\mathbf{c}})$ follows.*
3. *If $\overline{\alpha}(\mathscr{C}_{\mathbf{c}})$ is compatible or consistent, then so is $\mathscr{C}_{\mathbf{c}}$.*

Proof: Statement 3 follows immediately from (14). Focus next on Statement 2. Since set abstraction $\widehat{\alpha}$ is monotonic with respect to set inclusion, we deduce that contract abstraction $\overline{\alpha}$ is monotonic for $\preceq_{\mathbf{c}/\mathbf{a}}$. Regarding Statement 1, $\alpha(M_{\mathbf{c}}) \models_{\mathbf{a}}^{\mathrm{M}} \overline{\alpha}(\mathscr{C}_{\mathbf{c}})$ means

[20] An *ideal* of $(\mathcal{X}, \sqsubseteq)$ is a $\sqsubseteq$-downward closed subset of $\mathcal{X}$.

that $\gamma(\alpha(M_{\mathbf{c}})) \in \mathcal{M}_{\mathscr{C}_{\mathbf{c}}}$. By (11), $M_{\mathbf{c}} \sqsubseteq_{\mathbf{c}} \gamma(\alpha(M_{\mathbf{c}}))$, which, by Assumption 3, implies $M_{\mathbf{c}} \in \mathcal{M}_{\mathscr{C}_{\mathbf{c}}}$, i.e., $M_{\mathbf{c}} \models^{\mathrm{M}}_{\mathbf{c}} \mathscr{C}_{\mathbf{c}}$. Similarly, $\alpha(E_{\mathbf{c}}) \models^{\mathrm{E}}_{\mathbf{a}} \overline{\alpha}(\mathscr{C}_{\mathbf{c}})$ means that $\gamma(\alpha(E_{\mathbf{c}})) \in \mathcal{E}_{\mathscr{C}_{\mathbf{c}}}$. By (11), $E_{\mathbf{c}} \sqsubseteq_{\mathbf{c}} \gamma(\alpha(E_{\mathbf{c}}))$, which, by Assumption 3, implies $E_{\mathbf{c}} \in \mathcal{E}_{\mathscr{C}_{\mathbf{c}}}$, i.e., $E_{\mathbf{c}} \models^{\mathrm{E}}_{\mathbf{c}} \mathscr{C}_{\mathbf{c}}$.

Observe that Statement 1 allows proving implementation and environment relations based on abstractions. Similarly, Statement 3 allows proving compatibility or consistency based on abstractions. In contrast, Statement 2 allows disproving refinement based on abstractions.

The second part of our agenda is about compositionality of abstraction, with respect to both conjunction and parallel composition. Observe first that Statement 2 of Theorem 1 implies $\overline{\alpha}(\mathscr{C}^1_{\mathbf{c}} \wedge \mathscr{C}^2_{\mathbf{c}}) \preceq_{\mathbf{a}} \overline{\alpha}(\mathscr{C}^1_{\mathbf{c}}) \wedge \overline{\alpha}(\mathscr{C}^2_{\mathbf{c}})$, etc. Using, however, the fact that abstraction and concretizations for powersets arise from inverse maps, we can in fact get equalities:

Theorem 2 *The following equalities hold:*

$$\overline{\alpha}(\mathscr{C}^1_{\mathbf{c}} \wedge \mathscr{C}^2_{\mathbf{c}}) \quad = \quad \overline{\alpha}(\mathscr{C}^1_{\mathbf{c}}) \wedge \overline{\alpha}(\mathscr{C}^2_{\mathbf{c}}) \tag{16}$$

Proof: By definition,

$$\begin{aligned}
\overline{\alpha}(\mathscr{C}^1_{\mathbf{c}} \wedge \mathscr{C}^2_{\mathbf{c}}) &= \left(\widehat{\alpha}(\mathcal{E}_{(\mathscr{C}^1_{\mathbf{c}})} \cup \mathcal{E}_{(\mathscr{C}^2_{\mathbf{c}})}), \widehat{\alpha}(\mathcal{M}_{(\mathscr{C}^1_{\mathbf{c}})} \cap \mathcal{M}_{(\mathscr{C}^2_{\mathbf{c}})})\right) \\
\text{(by (15))} &= \left(\gamma^{-1}(\mathcal{E}_{(\mathscr{C}^1_{\mathbf{c}})} \cup \mathcal{E}_{(\mathscr{C}^2_{\mathbf{c}})}), \gamma^{-1}(\mathcal{M}_{(\mathscr{C}^1_{\mathbf{c}})} \cap \mathcal{M}_{(\mathscr{C}^2_{\mathbf{c}})})\right) \\
\text{(by (12))} &= \left(\gamma^{-1}(\mathcal{E}_{(\mathscr{C}^1_{\mathbf{c}})}) \cup \gamma^{-1}(\mathcal{E}_{(\mathscr{C}^2_{\mathbf{c}})}), \gamma^{-1}(\mathcal{M}_{(\mathscr{C}^1_{\mathbf{c}})}) \cap \gamma^{-1}(\mathcal{M}_{(\mathscr{C}^2_{\mathbf{c}})})\right) \\
&= \overline{\alpha}(\mathscr{C}^1_{\mathbf{c}}) \wedge \overline{\alpha}(\mathscr{C}^2_{\mathbf{c}})
\end{aligned}$$

which finishes the proof.

The last property in our agenda concerns parallel composition of contracts. We wish to relate $\overline{\alpha}(\mathscr{C}^1_{\mathbf{c}}) \otimes \overline{\alpha}(\mathscr{C}^2_{\mathbf{c}})$ and $\overline{\alpha}(\mathscr{C}^1_{\mathbf{c}} \otimes \mathscr{C}^2_{\mathbf{c}})$. Unlike previous properties, this does not come for free. We first need an additional property for the concretization of components: γ is called *sub-multiplicative* if

$$\gamma(X^1_{\mathbf{a}} \times_{\mathbf{a}} X^2_{\mathbf{a}}) \quad \sqsubseteq_{\mathbf{c}} \quad \gamma(X^1_{\mathbf{a}}) \times_{\mathbf{c}} \gamma(X^2_{\mathbf{a}}) \tag{17}$$

and *multiplicative* if equality holds in (17).

Theorem 3

1. *If γ is sub-multiplicative, then*

$$\overline{\alpha}(\mathscr{C}^1_{\mathbf{c}}) \otimes \overline{\alpha}(\mathscr{C}^2_{\mathbf{c}}) \quad \preceq_{\mathbf{a}} \quad \overline{\alpha}(\mathscr{C}^1_{\mathbf{c}} \otimes \mathscr{C}^2_{\mathbf{c}}) \tag{18}$$

2. *If, in addition, γ is multiplicative, then the two contracts $\overline{\alpha}(\mathscr{C}^1_{\mathbf{c}}) \otimes \overline{\alpha}(\mathscr{C}^2_{\mathbf{c}})$ and $\overline{\alpha}(\mathscr{C}^1_{\mathbf{c}} \otimes \mathscr{C}^2_{\mathbf{c}})$ possess identical sets of implementations—their sets of valid environments, however, may differ.*

3. *If, in addition to the condition stated in 2), $\mathscr{C}^1_{\mathbf{c}}$ and $\mathscr{C}^2_{\mathbf{c}}$ satisfy the following propery: $X_{\mathbf{c}} \models^{\mathrm{M}}_{\mathbf{c}} \mathscr{C}^i_{\mathbf{c}} \implies \gamma \circ \alpha(X_{\mathbf{c}}) \models^{\mathrm{M}}_{\mathbf{c}} \mathscr{C}^i_{\mathbf{c}}$, then, equality holds in (18).*

Proof: This is a difficult result and we refer the interested reader to [33].

3.7.1 Concluding discussion regarding contract abstraction

- From having a Galois connection on components we inherit an abstraction on contracts that is monotonic with respect to the refinement orders. Consistency and compatibility can both be checked on abstractions, see Theorem 1. The reader may conjecture that it should be possible to construct a Galois connection for contracts. We are rather convinced that this is not achievable, see [33] regarding obstructions.

- Theorem 2 allows checking consistency and compatibility in a $\wedge$-modular way by using equality

$$\overline{\alpha}(\bigwedge_{i\in I} \mathscr{C}^i_{\mathbf{c}}) \quad = \quad \bigwedge_{i\in I} \overline{\alpha}(\mathscr{C}^i_{\mathbf{c}})$$

- If γ is sub-multiplicative, Theorem 3 allows checking consistency in a $\otimes$-modular way by using refinement

$$\bigotimes_{i\in I} \overline{\alpha}(\mathscr{C}^i_{\mathbf{c}}) \quad \preceq_{\mathbf{a}} \quad \overline{\alpha}(\bigotimes_{i\in I} \mathscr{C}^i_{\mathbf{c}})$$

  This inequality is in the wrong way, however, for checking compatibility in a $\otimes$-modular way. Regarding this theorem, Galois connections on components where concretization is multiplicative are quite natural. Thus, Properties 1) and 2) of Theorem 3 will be easy to have. In contrast, the special condition, needed to have Property 3), arises only in exceptional cases. We conjecture that Theorem 3 is the best achievable result regarding compositionality of abstraction with respect to $\otimes$.

3.8 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. [19] and Chen et al. [66], albeit with deeply different and complementary approaches.

The publication [19] 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 4 are pairs of assertions. Sets of valid environments and sets of implementations are associated to contracts. Finally modal contracts are defined by borrowing ideas from modal specifications we discuss in Section 5. 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 [66] develops the concept of *declarative specification*, which consists of a tuple $\mathcal{P} = (\Sigma^{\text{in}}, \Sigma^{\text{out}}, T_\Sigma, F_\Sigma)$, where Σ^{in} and Σ^{out} are input and output alphabets of actions, $\Sigma = \Sigma^{\text{in}} \uplus \Sigma^{\text{out}}$, and $T_\Sigma, F_\Sigma \subseteq \Sigma^*$ such that $F_\Sigma \subseteq T_\Sigma$ are sets of *permissible* and *inconsistent* traces, respectively—this approach find its origins in earlier work

by Dill [94] and Negulescu [160]. 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. In the same direction, an algebraic theory of interface automata is proposed in [67], paying special attention to issues of safety (which is usual) and progress (which is not usual). Finally, [185] proposes a mathematical basis for multi-view modeling and [170] was an early paper proposing a notion of quotient for an interface model.

As far as we know, no notion of *abstraction* existed for contracts or specifications, with the attempt of being compliant with contract relations and operators. Our proposal here is new.

Observers: Observers, being related to the wide area of software and system testing, have been widely studied. A number of existing technologies support the design of observers and we review some of them now.

First of all, observers are related to the widely explored area of so-called IOCO-testing. The work [75] bridges the gap between this area and observers for contracts by re-considering compositional testing in view of contract composition.

Synchronous languages [26, 114, 32] are a formalism of choice in dealing with observers. The family of Synchronous Languages comprises mainly the imperative language Esterel [106, 97] and the dataflow languages Lustre [161] and Signal [173]. 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[21]. 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[22] 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 2.1). The widely used Simulink/Stateflow[23] tool by The Mathworks offers similar features. One slight drawback is that its mathematical semantics is less firmly defined (indeed, results of executions may differ depending on the code executed: simulation or generated C code). On the other hand, Simulink supports continuous time dynamics in the form of systems of interconnected ODEs (Ordinary Differential Equations), thus supporting the modeling of the physical part of the system. Using Simulink,

[21] http://www.esterel-technologies.com/products/scade-suite/
[22] http://www.geensoft.com/en/article/rtbuilder
[23] http://www.mathworks.com/products/simulink/

possibly enhanced with SimScape,[24] allows for including physical system models in observers, e.g., as part of the system environment. The same comment holds regarding Modelica.[25] Actually, observers have been proposed and advocated in the context of Lustre and Scade [115, 116, 117], Esterel [48], and Signal [153, 154]. 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 [167, 99, 98] 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 [4] 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 [15, 16]. 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 [174, 45], that is supported by the tool RATSY [46]. 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 [100]. A survey of using PSL in runtime verification can be found in [98].

4 Specializing to Assume/Guarantee contracts

Our static example of Section 2 provided an example of contract specified using Assumptions and Guarantees. In Assume/Guarantee contracts (A/G contracts), Assump-

[24] http://www.mathworks.com/products/simscape/
[25] https://www.modelica.org/

tions 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.

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) [129] and synchronous frameworks in which behaviors are sequences of successive reactions assigning values to the set of variables of the considered system. We first develop the theory for the simplest case of a fixed alphabet of variables. Then, we develop the other cases.

4.1 Dataflow A/G contracts

For this simplest variant, all components and contracts involve the same alphabet of variables V of variables, possessing identical[26] domain D. The restriction to a single alphabet of variables in this section is intended to simplify the concepts and the formulas. We will deal with variable alphabets later, in Section 4.3.

Assertions constitute the basis of A/G-components and contracts. An *assertion* is a set of behaviors:

$$P \quad \subseteq \quad V \mapsto \; D^* \cup D^\omega \tag{19}$$

i.e., a subset of the set of all finite or infinite behaviors over alphabet of variables V. Assertions are equipped with the boolean algebra $\cap, \cup, \neg$, where the latter denotes set complement.

Behaviors are generically denoted by the symbol σ; a behavior associates, to each symbol $x \in V$, 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 4.4 variants of this framework with synchronous models of behaviors. For $V' \subset V$ and σ a behavior, $\mathbf{pr}_{V'}(\sigma)$ is the restriction of σ to the sub-alphabet V'. We simply write $\mathbf{pr}_x(\sigma)$ if $V' = \{x\}$. Behaviors are partially ordered by the *prefix* relation denoted by $\sigma' \sqsubseteq \sigma$, meaning that, for every $x \in V$, the word $\sigma'(x)$ is a prefix of the word $\sigma(x)$. We denote by ϵ the empty word of D^*.

Definition 3 *A* component *is a non-empty and prefix-closed assertion. The component doing nothing is modeled by the singleton* $\{\epsilon^V\}$*. For* P *an arbitrary assertion, denote by* $P^\downarrow$ *the maximal prefix-closed subset of* P*. If* $P^\downarrow$ *is non-empty, then it is the maximal component contained in* P*.*

Despite the fact that a component is typically implemented in practice in form of a program, we intentionally define it in Definition 3 as an abstract assertion. This definition gives us greater flexibility and does not enforce any particular syntax. The abstract assertions are, however, not effective and their syntax must be fixed in order to allow finite description of the component behavior. The choice of the syntax is crucial and affects both the expressiveness and succintness of the assertion language. Once the

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

semantic domain of the component is defined, both abstract assertions and concrete implementations share a common feature—they define the component behavior in terms of traces.

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

$$P_1 \times P_2 \quad = \quad P_1 \cap P_2 \tag{20}$$

Formulas (19) and (20) 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 4.2 and components are input/output *functions*, we obtain the model of *Kahn Process Networks* [129] widely used for the mapping of synchronous programs over distributed architectures [176, 177].

Definition 4 *A* contract *is a pair* $\mathscr{C} = (A, G)$ *of assertions, called the* assumptions *and the* guarantees. *The set* $\mathcal{E}_{\mathscr{C}}$ *of the legal environments for* $\mathscr{C}$ *collects all components* E *such that* $E \subseteq A$. *The set* $\mathcal{M}_{\mathscr{C}}$ *of all components implementing* $\mathscr{C}$ *is defined by* $A \times M \subseteq G$.

Observe that we are not requiring any particular condition on the sets A and G. They can be empty and need not be prefix-closed. By Definition 3, contract $\mathscr{C} = (A, G)$ is *compatible* if and only if $A^{\downarrow}$ is nonempty and we denote by $E_{\mathscr{C}} = A^{\downarrow}$ the maximal (for set inclusion) environment of $\mathscr{C}$. Denoting by $\neg A$ the complement of set A, any component M such that $M \subseteq G \cup \neg A$ is an implementation of $\mathscr{C}$. Thus, contract $\mathscr{C} = (A, G)$ is *consistent* if and only $(G \cup \neg A)^{\downarrow}$ is nonempty and we denote by $M_{\mathscr{C}} = (G \cup \neg A)^{\downarrow}$ the maximal (for set inclusion) implementation of $\mathscr{C}$. Observe that two contracts $\mathscr{C}$ and $\mathscr{C}'$ with identical alphabets of variables, identical assumptions, and such that $(G' \cup \neg A')^{\downarrow} = (G \cup \neg A)^{\downarrow}$, possess identical sets of implementations: $\mathcal{M}_{\mathscr{C}} = \mathcal{M}_{\mathscr{C}'}$. According to our meta-theory, such two contracts are equivalent. Say that contract

$$\mathscr{C} = (A, G) \text{ is } \textit{saturated} \text{ if } A = A^{\downarrow}, G = (G \cup \neg A)^{\downarrow}.$$

Contract $\mathscr{C} = (A, G)$ is equivalent to its saturated form $(A^{\downarrow}, (G \cup \neg A)^{\downarrow})$. Next, for $\mathscr{C}$ and $\mathscr{C}'$ two saturated contracts with identical alphabets of variables,

$$\textit{refinement } \mathscr{C}' \preceq \mathscr{C} \text{ holds in the sense of the meta-theory iff } \left\{ \begin{array}{lcl} A' & \supseteq & A \\ G' & \subseteq & G \end{array} \right. \tag{21}$$

As a consequence of (21), Assumptions 1 and 2 of the meta-theory hold for A/G contracts. *Conjunction* follows from the refinement relation: for $\mathscr{C}_1$ and $\mathscr{C}_2$ two saturated contracts with identical alphabets of variables:

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

Focus now on contract *composition* $\mathscr{C} = \mathscr{C}_1 \otimes \mathscr{C}_2$.

Lemma 4 *For saturated contracts, contract composition instantiates through the following formulas:*

$$G = G_1 \cap G_2 \quad \textit{and} \quad A = \max \left\{ A \;\middle|\; \begin{array}{ll} A = A^{\downarrow} & \text{and} \\ A \cap G_2 \subseteq A_1 & \text{and} \\ A \cap G_1 \subseteq A_2 & \end{array} \right\} \tag{22}$$

Proof: We use the notation $\mathbf{C}_{\mathscr{C}_1,\mathscr{C}_2}$ introduced in Section 3.4. It is enough to prove the following:

$$\mathbf{C}_{\mathscr{C}_1,\mathscr{C}_2} = \left\{ (A,G) \;\middle|\; \begin{array}{ll} (A,G) \text{ saturated} & \text{and} \\ A \cap G_2 \subseteq A_1 & \text{and} \\ A \cap G_1 \subseteq A_2 & \end{array} \right\} \tag{23}$$

To prove inclusion $\supseteq$ in (23), pick M_1, M_2, and E such that $E \subseteq A$, $M_1 \subseteq G_1$, and $M_2 \subseteq G_2$. Then, $M_1 \times M_2 = M_1 \cap M_2 \subseteq G$, whence $M_1 \times M_2 \models^{\mathrm{M}} (A,G)$ follows. Then, $E \times M_2 = E \cap M_2 \subseteq A \cap G_2 \subseteq A_1$. This proves that $E \times M_2 \models^{\mathrm{E}} (A_1, G_1)$, and the inclusion $\supseteq$ in (23) follows. To prove inclusion $\subseteq$ in (23), pick a saturated pair (A,G) belonging to $\mathbf{C}_{\mathscr{C}_1,\mathscr{C}_2}$ and take $E = A$, $M_1 = G_1$, and $M_2 = G_2$. By definition of $\mathbf{C}_{\mathscr{C}_1,\mathscr{C}_2}$, we get $M_1 \times M_2 \models^{\mathrm{M}} (A,G)$, $M_2 \times E \models^{\mathrm{E}} (A_1, G_1)$, and $M_1 \times E \models^{\mathrm{E}} (A_2, G_2)$, which translates as $G_1 \cap G_2 \subseteq G$, $G_2 \cap A \subseteq A_1$, and $G_1 \cap A \subseteq A_2$. □

Formula (22) satisfies Assumption 2 of the meta-theory. Variational formulas (22) reformulate as the formulas originally proposed in [28]:

$$G = G_1 \cap G_2 \qquad \text{and} \qquad A = (A_1 \cap A_2) \cup (\neg(G_1 \cap G_2))^{\downarrow} \tag{24}$$

Observe that the so obtained contract (A,G) is indeed saturated.

No quotient operation is known for Assume/Guarantee contracts.

As the reader has noticed, getting saturated contracts is important in applying this contract algebra. This seems to require being able to compute with unions and complements of assertions. In fact, we only need to be able to compute the operation $A \cup \neg G$, which we like to interpret as the entailment $A \Rightarrow G$. Thus it is enough to have a tool able to synthesize models for formulas of the form $A \Rightarrow G$, where: G is a formula or a conjunction of formulas, A is a formula or a conjunction of formulas, or, recursively, A has the form $A \Rightarrow G$.

We finish this section by observing that the two contracts $\mathscr{C}_1$ and $\mathscr{C}_1'$ of Section 2.2.1 satisfy $\mathscr{C}_1' \preceq \mathscr{C}_1$ according to the theory of this section: guarantees are identical but assumptions are relaxed.

4.2 Capturing exceptions

Referring to the primer of Section 2.2.1 and its static system example, the contract $\mathscr{C}_1$ under-specifies the case when $y = 0$, modeling the assumption that the environment never provides this input. If for some reason this input is nevertheless given to a component that implements $\mathscr{C}_1$, the component has full freedom on how to react to this input. In particular, if the component decides to compute $z = x/y$ even when $y = 0$, it can lead to an upredictable outcome and possibly a crash unless exeption 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 of variables V. 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 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 $M = (V^{\mathrm{in}}, V^{\mathrm{out}}, P)$, where $V = V^{\mathrm{in}} \cup V^{\mathrm{out}}$ is the decomposition of alphabet of variables V into its *inputs* and *outputs*, and $P \subseteq (V \mapsto (D^* \cup D^\omega))$

is a non-empty prefix-closed assertion. Whenever convenient, we shall denote by V_M^{in}, Q_M, P_M, etc., the items defining component M. Components M_1 and M_2 are *composable* if $V_1^{\text{out}} \cap V_2^{\text{out}} = \emptyset$. If so, then $M_1 \times M_2 = (V^{\text{in}}, V^{\text{out}}, P)$ is well defined, with $V^{\text{out}} = V_1^{\text{out}} \cup V_2^{\text{out}}$, $V^{\text{in}} = V - V^{\text{out}}$, 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 2, 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 5 *A component M is* free of exception *if:*

1. *It accepts all inputs:*

$$\mathbf{pr}_{V_M^{\text{in}}}(P_M) = V_M^{\text{in}} \mapsto (D^* \cup D^\omega)$$

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

3. *Status "fail" is persistent: for all behaviors $\sigma \in P_M$, for all $x \in V_M$, if $\sigma' = \mathbf{pr}_x(\sigma)$ can be decomposed into $\sigma'_1 \cdot fail \cdot \sigma'_2$, then $\sigma'_2 = fail^\omega$.*

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 5 defines our family of components. A/G contracts are re-defined accordingly:

Definition 6 *A* contract *is a tuple $\mathscr{C} = (V^{\text{in}}, V^{\text{out}}, A, G)$, where V^{in} and V^{out} are the input and output alphabets of variables and A and G are assertions over V, called the* assumptions *and the* guarantees. *The set $\mathcal{E}_{\mathscr{C}}$ of the legal environments for $\mathscr{C}$ are all free of exception components E such that $V_E^{\text{in}} = V_{\mathscr{C}}^{\text{out}}$, $V_E^{\text{out}} = V_{\mathscr{C}}^{\text{in}}$, and $P_E \subseteq A$. The set $\mathcal{M}_{\mathscr{C}}$ of all components implementing $\mathscr{C}$ is defined by: M is free of exception, $V_M^{\text{in}} = V_{\mathscr{C}}^{\text{in}}$ and $V_M^{\text{out}} = V_{\mathscr{C}}^{\text{out}}$ and $P_{E \times M} \subseteq G$ for every environment E of $\mathscr{C}$.*

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

Property 7 *Let $\mathscr{C} = (V^{\text{in}}, V^{\text{out}}, A, G)$ be a contract satisfying the following conditions:*

$$\mathbf{pr}_{V_{\mathscr{C}}^{\text{in}}}(G) = V_M^{\text{in}} \mapsto (D^* \cup D^\omega) \tag{25}$$

$$\mathbf{pr}_{V_{\mathscr{C}}^{\text{out}}}(A) = V_M^{\text{out}} \mapsto (D^* \cup D^\omega) \tag{26}$$

$$\text{"fail" does not occur in } G \cap A \tag{27}$$

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

Proof: By condition (25), the component $M = (V^{\text{in}}, V^{\text{out}}, G)$ satisfies condition 1 of Definition 5. It may not satisfy conditions 2 nor 3, however. To achieve this we modify, in M, the behaviors not belonging to A in order to avoid *fail* to occur (preserving M on A will ensure that the modification still implements $\mathscr{C}$). To get the desired modification M', replace any behavior σ of M belonging to $G \cap \neg A$ by a behavior σ' such that

$\mathbf{pr}_{V^{\text{in}}_{\mathscr{C}}}(\sigma') = \mathbf{pr}_{V^{\text{in}}_{\mathscr{C}}}(\sigma)$ and $\mathbf{pr}_{V^{\text{out}}_{\mathscr{C}}}(\sigma') \notin \mathit{fail}$. Component M' obtained in this way is free of exception and implements $\mathscr{C}$, showing that $\mathscr{C}$ is consistent.

Consider next the component $E = (V^{\text{out}}, V^{\text{in}}, A)$. If E is free of exception, then it is an environment for $\mathscr{C}$. If this is not the case, then we can modify E on $A \cap \neg G$ as we did for obtaining M', thus obtaining a modification E' that is free of exception and still satisfies $E' \subseteq A$. Thus, E' is a legal environment for $\mathscr{C}$, showing that $\mathscr{C}$ is compatible. □

Conditions (25) and (26) express that assumptions and guarantees are both input enabled. Condition (27) is the key one. Observe that, now, contract $\mathscr{C}'_1$ of Section 2.2.1 is inconsistent since it has no implementation—implementations must be free of exception. In turn, contract $\mathscr{C}_1$ is consistent. This is in contrast to the theory of Section 4.1, where both contracts were considered consistent (crashes were not ruled out). Indeed, contract $\mathscr{C}_1$ of Section 2.2.1 satisfies the conditions of Property 7, whereas $\mathscr{C}'_1$ does not. Addressing exceptions matters.

4.3 Dealing with variable alphabets

Since contracts aim at capturing incomplete designs, we cannot restrict ourselves to a fixed alphabet of variables—it is not known in advance what the actual alphabet of variables of the complete design will be. Thus the simple variants of Sections 4.1 and 4.2 have no practical relevance and we must extend them to dealing with variable alphabets. In particular, components will now be pairs $M = (V_M, P_M)$, where V_M is the alphabet of variables of M and P_M is an assertion over V_M. Similarly, contracts are tuples $\mathscr{C} = ((V_A, A), (V_G, G))$, where assumptions A and guarantees G are assertions over alphabets of variables V_A and V_G.

Key to dealing with variable alphabet of variables is the operation of *alphabet equalization* that we introduce next. For P an assertion over alphabet of variables V and $V' \subseteq V$, we consider its *projection* $\mathbf{pr}_{V'}(P)$ over V', which is simply the set of all restrictions, to V', of all behaviors belonging to P. We will also need the *inverse projection* $\mathbf{pr}^{-1}_{V''}(P)$, for $V'' \supseteq V$, which is the set of all behaviors over V'' projecting to V as behaviors of P. For $(V_i, P_i), i = 1, 2$, we call *alphabet equalization* of (V_1, P_1) and (V_2, P_2) the two assertions $(V, \mathbf{pr}^{-1}_V(P_i)), i = 1, 2$ where $V = V_1 \cup V_2$. We also need to define alphabet equalization when the alphabet of variables V decomposes as $V = V^{\text{in}} \uplus V^{\text{out}}$. Equalizing the above decomposition to a larger alphabet of variables $V'' \supseteq V$ yields $V''^{\text{out}} = V^{\text{out}}$, whence $V''^{\text{in}} = V'' \setminus V''^{\text{out}}$ follows. Alphabet equalization serves as a preparation step to reuse the framework of Section 4.1 when alphabets are variable.

This being defined, all operations and relations introduced in Section 4.1 are extended to the case of variable alphabet by 1) applying alphabet equalization to the involved assertions, and, 2) reusing the operation or relation as introduced in Section 4.1.

A practical pitfall of A/G contracts with variable alphabet: As pointed out in [28], this generalization yields a contract theory that is a valid instantiation of the meta-theory (up to the missing quotient). It is not fully satisfactory from the practical standpoint, however. The reason is that the conjunction of two contracts with disjoint alphabets of variables yields a trivial assumption T, which is very demanding—any environment must be accommodated—and does not reflect the intuition. This will be further discussed in Section 5.4. □

To summarize, Sections 4.1, 4.2, and 4.3, together define a framework of *asyn-*

chronous Assume/Guarantee contracts where components are of Kahn Process Network type. In the next section, we sketch a variant where components are synchronous transition systems.

4.4 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

$$P \quad \subseteq \quad (V \mapsto D)^* \cup (V \mapsto D)^\omega. \tag{28}$$

Compare (28) with (19). In both cases, assertions are sets of behaviors. With reference to (19), behaviors were tuples of finite or infinite flows, one for each symbol of the alphabet of variables. In contrast, in (28), behaviors are finite or infinite sequences of *reactions*, which are the assignment of a value to each symbol of the alphabet of variables. 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 [32]. Definition (28) for assertions correspond to the synchronous model of computation, whereas (19) corresponds to the Kahn Process Network type of model [129, 159]. The material of Sections 4.1, 4.2, and 4.3, can be adapted to this new model of component composition, thus yielding a framework of *synchronous Assume/Guarantee contracts.*

4.5 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 4.1 and 4.2. Let P be an assertion according to (19). P defines a verdict b_P by setting

$$b_P(\sigma) = \text{T} \text{ if and only if } \sigma \in P \tag{29}$$

Observers must return their verdict in some finite amount of time. Hence, an online interpretation of Definition 4 is appropriate. With this interpretation, for $\mathscr{C} = (V^{\text{in}}, V^{\text{out}}, A, G)$ a contract, its associated observer is obtained as follows, with reference to Definition 2:

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

Lemma 2 for generic observers specializes to the following, effective, semi-decision procedure:

Lemma 5

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

4.6 Abstractions

For concepts and undefined notations, the reader is referred to Section 3.7. We assume two sets $\mathbb{M}_{\mathbf{c}}$ and $\mathbb{M}_{\mathbf{a}}$ of concrete and abstract components. $\mathbb{M}_{\mathbf{c}}$ and $\mathbb{M}_{\mathbf{a}}$ are ordered by set inclusion. We assume a Galois connection (α, γ) between $\mathbb{M}_{\mathbf{c}}$ and $\mathbb{M}_{\mathbf{a}}$. Concrete and abstract A/G contracts are pairs $\mathscr{C}_{\mathbf{c}} = (A_{\mathbf{c}}, G_{\mathbf{c}})$ and $\mathscr{C}_{\mathbf{a}} = (A_{\mathbf{a}}, G_{\mathbf{a}})$ of concrete/abstract assumptions and guarantees. $E_{\mathbf{c}} \models^{\mathrm{E}}_{\mathbf{c}} \mathscr{C}_{\mathbf{c}}$ iff $E_{\mathbf{c}}$ satisfies $A_{\mathbf{c}}$, i.e., $E_{\mathbf{c}} \subseteq A_{\mathbf{c}}$. $M_{\mathbf{c}} \models^{\mathrm{M}}_{\mathbf{c}} \mathscr{C}_{\mathbf{c}}$ iff $M_{\mathbf{c}} \cap A_{\mathbf{c}}$ satisfies $G_{\mathbf{c}}$, i.e., $M_{\mathbf{c}} \cap A_{\mathbf{c}} \subseteq G_{\mathbf{c}}$. Since components are sets of behaviors, the intuitive choice for $\sqsubseteq_{\mathbf{c/a}}$ is set inclusion. This complies with Assumption 3. Using (15) yields, for two concrete and abstract A/G contracts $\mathscr{C}_{\mathbf{c}} = (A_{\mathbf{c}}, G_{\mathbf{c}})$ and $\mathscr{C}_{\mathbf{a}} = (A_{\mathbf{a}}, G_{\mathbf{a}})$, $\overline{\alpha}(\mathscr{C}_{\mathbf{c}}) = (\widehat{\alpha}(\mathcal{E}_{\mathscr{C}_{\mathbf{c}}}), \widehat{\alpha}(\mathcal{M}_{\mathscr{C}_{\mathbf{c}}}))$, where:

$$\begin{array}{rcl} \widehat{\alpha}(\mathcal{E}_{\mathscr{C}_{\mathbf{c}}}) & = & \gamma^{-1}(\mathcal{E}_{\mathscr{C}_{\mathbf{c}}}) \\ & = & \{E_{\mathbf{a}} \mid \gamma(E_{\mathbf{a}}) \subseteq A_{\mathbf{c}}\} \\ \text{(by using (10))} & = & \{E_{\mathbf{a}} \mid E_{\mathbf{a}} \subseteq \alpha(A_{\mathbf{c}})\} \end{array}$$

and

$$\begin{array}{rcl} \widehat{\alpha}(\mathcal{M}_{\mathscr{C}_{\mathbf{c}}}) & = & \gamma^{-1}(\mathcal{M}_{\mathscr{C}_{\mathbf{c}}}) \\ & = & \{M_{\mathbf{a}} \mid \gamma(M_{\mathbf{a}}) \cap A_{\mathbf{c}} \subseteq G_{\mathbf{c}}\} \\ \text{(by using (10))} & = & \{M_{\mathbf{a}} \mid M_{\mathbf{a}} \subseteq \alpha(G_{\mathbf{c}} \cup \neg A_{\mathbf{c}})\} \end{array}$$

To summarize:

$$\overline{\alpha}(A_{\mathbf{c}}, G_{\mathbf{c}}) = (\alpha(A_{\mathbf{c}}), \alpha(G_{\mathbf{c}} \cup \neg A_{\mathbf{c}})) \tag{30}$$

It remains to explain how to construct a Galois connection for components defined as sets of behaviors. To be able to apply Theorem 3, we are interested in knowing if γ is (sub)-multiplicative. It is shown in [33] that predicate abstraction applied to both the initial condition and the transition relation, defines a Galois connection over components in which concretization γ is multiplicative, so that Statements 1 and 2 of Theorem 3 apply. Recall that predicate abstraction works as follows—we explain it for transition relations. Select an arbitrary finite set $(P_i)_{i \in \mathbf{I}}$ of predicates over $2^{D_{\mathbf{c}} \times D_{\mathbf{c}}}$. For a concrete transition relation $R \subseteq 2^{D_{\mathbf{c}} \times D_{\mathbf{c}}}$, $P_i(R)$ returns *true* or *false*, depending on whether R satisfies this predicate or not. The abstraction of R is then defined as $\alpha(R) = (P_i(R))_{i \in \mathbf{I}} \in \mathbf{Bool}^{\mathbf{I}} =_{\mathrm{def}} \mathbb{M}_{\mathbf{a}}$. It is seen that α is a complete $\sqcup_{\mathbf{c}}$-morphism, hence a unique concretization γ can be canonically associated with it, making (α, γ) a Galois connection [73, 157]. Equipping the abstract domain $\mathbb{M}_{\mathbf{a}}$ with the product $\times_{\mathbf{a}} =_{\mathrm{def}} \bigcap_{\mathbf{a}}$, where the infimum $\bigcap_{\mathbf{a}}$ refers to the product order on the abstract domain $\mathbb{M}_{\mathbf{a}} = \mathbf{Bool}^{\mathbf{I}}$, ensures that the so defined γ is multiplicative. See [33] for details.

4.7 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), does not handle variable alphabets in a satisfactory way. Also, it suffers from the need to manipulate negations of assertions, an operation that is generally not effective, except if the framework is restricted to boolean transition systems. For general frameworks, using observers or abstract interpretation can mitigate this in part.

4.8 Bibliographical note

By explicitly relying on the notions of Assumptions and Guarantees, A/G contracts are intuitive, which makes them appealing for the engineer. In A/G contracts, Assumptions and Guarantees are just properties. The typical case is when these properties are languages or sets of traces, which includes the class of safety properties [133, 65, 152, 14, 70]. A/G contracts were advocated by the SPEEDS project [28]. They were further experimented in the framework of the CESAR project [76]. The theory developed in [28] turns out to be closest to this presentation; still, exceptions were not handled in [28]. The presentation developed in this paper clarifies the design choices in A/G contract theories.

Interface Input/Output automata were proposed in [138] as a pair of two i/o-automata acting as assumption and guarantee. A comparison with interface automata is given.

Inspired by [138], another form for A/G contract was proposed by [110, 111, 113] when designs are expressed using the BIP programming language [44, 195]. To achieve separate development of components, and to overcome the problems that certain models have with the effective computation of the operators, the authors avoid using parallel composition $\otimes$ of contracts. Instead, they replace it with the concept of *circular reasoning*, which states as follows in its simplest form: if design M satisfies property G under assumption A and if design N satisfies assumption A, then $M \times N$ satisfies G. When circular reasoning is sound, it is possible to check relations between composite contracts based on their components only, without taking expensive compositions. In order for circular reasoning to hold, the authors devise restricted notions of refinement under context and show how to implement the relations in the contract theory for the BIP framework. Compatibility is not addressed and this proposal does not consider conjunction.

A/G contracts are proposed in [68] for finite traces of interface automata. Safety and progress for possibly nondeterministic automata are addressed by characterizing a component through *observable, inconsistent* (raising an exception), and *quiescent* (reaction termination) traces. The satisfaction relations for environments and implementations are adjusted to account for this more precise characterization of components. Refinement, conjunction, disjunction, parallel composition, and quotient are proposed: this development is therefore remarkably comprehensive. An interesting comparision with [138] is developed using an illustration example.

The automatic generation of observers for A/G contracts has been proposed in the work [103, 102], where assertions are specified using a declarative pattern-based language. A set of monitors is then generated and implemented in the Simulink framework to observe the underlying system execution and flag behaviors that violate either the assumptions or the guarantees. The method seems suitable for analyzing the implementation relation, while consistency and compatibility are only analyzed for closed systems.

Regarding extensions, a notion of *contract for real-time interfaces* is proposed in [43]. 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 the same direction, A/G contracts were proposed in [198, 186, 187, 197] for real-time scheduling problems, where tasks and their data dependencies, and resources, must be handled. See also Section 4 of companion paper [30].

In [162], 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. [87, 88, 89]. 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 many respects. For the first time, it is cast into the meta-theory of contracts, with the advantage of clarifying the definition of refinement and parallel composition of contracts—this involved some hand waving in the original work [28]. Also, this allowed us to handle exceptions. This presentation of A/G contracts is complemented by an application case in real-time scheduling in the context of AUTOSAR developed in Section 4 of companion paper [30].

5 Specializing to Interface theories

Interface theories are an interesting alternative to Assume/Guarantee contracts. They aim at providing a merged specification of the implementations and environments as-

sociated 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 [151, 150] as their framework for components and environments. As a prerequisite, we thus recall the background on Input/Output Automata, i/o-automata for short.

5.1 Components as i/o-automata

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

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

An i/o-automaton can be interpreted as an open system: the transitions labeled by actions in Σ^{out} represents the outputs that the system can generate while the transitions labeled by actions in Σ^{in} represent the inputs a system can accept. By concatenation, the transition relation $\rightarrow$ extends to a relation $\rightarrow^*$ on $Q \times \Sigma^* \times Q$, where Σ^* is the set of all finite words over Σ. Say that a state q' is *reachable* from q if there exists some word μ such that $q \xrightarrow{\mu} {}^* q'$. To considerably simplify the development of the theory, we restrict ourselves to

$$\textit{deterministic} \text{ i/o-automata, i.e.:} \quad [q \xrightarrow{\alpha} q_1 \text{ and } q \xrightarrow{\alpha} q_2] \implies q_2 = q_1 \tag{31}$$

and we denote by

$$\alpha \mapsto \delta(q, \alpha) \tag{32}$$

the partial function such that $\delta(q, \alpha)$ is the unique (if it exists) state such that $q \xrightarrow{\alpha} \delta(q, \alpha)$.

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

$$\Sigma^{\text{out}} = \Sigma_1^{\text{out}} \cup \Sigma_2^{\text{out}} \; ; \; \Sigma^{\text{in}} = \left(\Sigma_1^{\text{in}} \cup \Sigma_2^{\text{in}}\right) \setminus \Sigma^{\text{out}} \; ; \; Q = Q_1 \times Q_2 \text{ and } q_0 = (q_{1,0}, q_{2,0}),$$

and the transition relation $\rightarrow$ is given by

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

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 $q_2 \preceq q_1$ if

$$\forall \alpha, q_2' \text{ such that } q_2 \xrightarrow{\alpha}_2 q_2' \quad \implies \quad \exists q_1' \text{ such that } [q_1 \xrightarrow{\alpha}_1 q_1' \text{ and } q_2' \preceq q_1'] \tag{33}$$

Say that

$$M_1 \textit{ simulates } M_2, \text{ written } M_2 \preceq M_1, \text{ if } q_{2,0} \preceq q_{1,0}. \tag{34}$$

Observe that simulation relation (33,34) does not distinguish inputs from outputs neither it distinguishes the component from its environment. It is the classical simulation relation meant for closed systems.

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

$$M^{\uparrow\Sigma'} \quad = \quad \left(\Sigma^{\text{in}} \cup (\Sigma' \setminus \Sigma), \Sigma^{\text{out}}, Q, q_0, \rightarrow'\right)$$

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 Environments are receptive i/o-automata: 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:[27]

$$M \text{ is } \textit{receptive} \text{ iff } \forall q \in Q, \forall \alpha \in \Sigma^{\text{in}} : \; q \xrightarrow{\alpha} \tag{35}$$

Receptiveness is stable under parallel composition.

The following simple technique can be used to make i/o-automaton M receptive: augment Q with the extra "top" state $\top$ and, for each pair $(q, \alpha) \in Q \times \Sigma^{\text{in}}$ such that α is not enabled at q, add a transition $q \xrightarrow{\alpha} \top$ and then add all self-loops $\top \xrightarrow{\alpha} \top$ for any action $\alpha \in \Sigma$. This yields a receptive i/o-automaton that we denote by

$$\overline{M} \tag{36}$$

and we recover M from $\overline{M}$ by removing the extra state $\top$ and all transitions leading to it.

5.2 Interface Automata with fixed alphabet

For reasons that will become clear later, we restrict the presentation of interface automata to the case of a fixed alphabet Σ. We begin with the definition of Interface Automata which are possibly non-receptive i/o-automata. Moreover we give their semantics as contracts, that is, as pairs formed of a component and of a valid environment.

Definition 7 *An* Interface Automaton *is a tuple* $\mathscr{C} = (\Sigma^{\text{in}}, \Sigma^{\text{out}}, Q, q_0, \rightarrow)$ *where* $\Sigma^{\text{in}}, \Sigma^{\text{out}}, Q,$ *and* $\rightarrow$ *are as in i/o-automata. The initial state* q_0*, however, may not satisfy* $q_0 \in Q$*. If* $q_0 \in Q$ *holds, Interface Automaton* $\mathscr{C}$ *defines a contract by fixing a pair* $(\mathcal{E}_{\mathscr{C}}, \mathcal{M}_{\mathscr{C}})$ *as follows:*

The set $\mathcal{E}_{\mathscr{C}}$ *of* legal environments *for* $\mathscr{C}$ *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* $\mathscr{C}$*, seen as i/o-automata, are composable;*
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* $E \times \mathscr{C}$*, then* $(q_E, q) \xrightarrow{\alpha}_{E\times\mathscr{C}}$ *holds.*

Now define the particular environment $E_{\mathscr{C}} \in \mathcal{E}_{\mathscr{C}}$*, such that* $E_{\mathscr{C}} \times \mathscr{C}$ *simulates* $E \times \mathscr{C}$ *in the sense of* (34) *for any* $E \in \mathcal{E}_{\mathscr{C}}$*; we define* $E_{\mathscr{C}}$ *as follows:*

[27] In fact, receptiveness is assumed in the original notion of i/o-automaton by Nancy Lynch [151, 150]. We use here a relaxed version of i/o-automaton for reasons that will become clear later.

(a) $E_{\mathscr{C}} = (\Sigma^{\text{out}}, \Sigma^{\text{in}}, Q \cup \{\top\}, q_0, \rightarrow_{E_{\mathscr{C}}})$*, where* $\top$ *is a special extra state;*

(b) transition relation $\rightarrow_{E_{\mathscr{C}}}$ *is such that its restriction to* $Q{\times}\Sigma{\times}Q$ *coincides with* $\rightarrow$*; and*

(c) we make the resulting i/o-automaton receptive as indicated in (36).

The set $\mathcal{M}_{\mathscr{C}}$ *of the* implementations *of* $\mathscr{C}$ *collects all components* M *such that i/o-automaton* $\mathscr{C}$ *simulates* $E_{\mathscr{C}}{\times}M$ *in the sense of* (34)*.*

Condition 2 means that environment E is only willing to emit an output if it is accepted as an input by $\mathscr{C}$ in the composition $E \times \mathscr{C}$.

Lemma 6 *The environment* $E_{\mathscr{C}}$ *is maximal (with respect to simulation) in* $\mathcal{E}_{\mathscr{C}}$*.*

Proof: Call $\check{E}_{\mathscr{C}}$ the i/o-automaton obtained by applying (a) and (b) but not (c). By construction, $\check{E}_{\mathscr{C}} \times \mathscr{C}$ is isomorphic with $(\emptyset, \Sigma, Q, q_0, \rightarrow)$, i.e., it is obtained from $\mathscr{C}$, seen as an i/o-automaton, by simply turning inputs to outputs. Consequently, Condition 2. holds for $\check{E}$ and this i/o-automaton is maximal (with respect to simulation) having this property. Step (c) preserves both Condition 2. and maximality when making $\check{E}$ receptive. □ The above definition of Interface Automata is heterodox, compare with the original references [84, 6]. Definition 7 introduces the two sets $\mathcal{E}_{\mathscr{C}}$ and $\mathcal{M}_{\mathscr{C}}$, 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 7 allows us to cast Interface Automata in the framework of the meta-theory of Table 2.

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

Lemma 7 $q_0 \in Q$ *is the necessary and sufficient condition for* $\mathscr{C}$ *to be both consistent and compatible in the sense of the meta-theory, i.e.,* $\mathcal{E}_{\mathscr{C}} \neq \emptyset$ *and* $\mathcal{M}_{\mathscr{C}} \neq \emptyset$*.*

Refinement and conjunction. Contract *refinement* as defined in Table 2 is equivalent to *alternating simulation* [11], defined as follows: for $\mathscr{C}_i, i = 1, 2$ two Interface Automata, say that two respective states $q_i, i = 1, 2$ are in alternating simulation, written $q_2 \preceq q_1$, if

$$\begin{array}{rcl} \forall \alpha \in \Sigma_2^{\text{out}}, q_2' \text{ s.t. } q_2 \xrightarrow{\alpha}_2 q_2' & \Rightarrow & \left\{ \begin{array}{l} \alpha \in \Sigma_1^{\text{out}}, \text{ and} \\ \exists q_1' \text{ s.t. } q_1 \xrightarrow{\alpha}_1 q_1' \text{ and } q_2' \preceq q_1' \end{array} \right. \\ \forall \alpha \in \Sigma_1^{\text{in}}, q_1' \text{ s.t. } q_1 \xrightarrow{\alpha}_1 q_1' & \Rightarrow & \left\{ \begin{array}{l} \alpha \in \Sigma_2^{\text{in}}, \text{ and} \\ \exists q_2' \text{ s.t. } q_2 \xrightarrow{\alpha}_2 q_2' \text{ and } q_2' \preceq q_1' \end{array} \right. \end{array} \tag{37}$$

Say that $\mathscr{C}_2$ *refines* $\mathscr{C}_1$, written $\mathscr{C}_2 \preceq \mathscr{C}_1$, if $q_{2,0} \preceq q_{1,0}$. The first condition of (37) reflects the inclusion $\mathcal{M}_{\mathscr{C}_2} \subseteq \mathcal{M}_{\mathscr{C}_1}$, whereas the second condition of (37) reflects the opposite inclusion $\mathcal{E}_{\mathscr{C}_2} \supseteq \mathcal{E}_{\mathscr{C}_1}$. As Interface Automata are taken deterministic, a matching state q_1' for q_2' (or q_2' for q_1') is unique when it exists. Alternating simulation can be effectively checked, see [82] for issues of computational complexity. We note that we use simulation and alternating simulation and refinement relations for components and contracts, respectively. It is sufficient to use the classical simulation relation between components because we assume that components are input-enabled. In fact,

for input-enabled systems, simulation and alternating simulation coincide. In addition, for deterministic (contracts) components, (alternating) simulation also coincides with (alternating) language inclusion.

Unfortunately, no simple formula for the conjunction of contracts is known. See [96] for the best results in this direction.

Parallel composition and quotient: Contract *composition* $\mathscr{C}_1 \otimes \mathscr{C}_2$, as defined in the meta-theory, is effectively computed as follows, for $\mathscr{C}_i$ two Interface Automata satisfying the conditions of Lemma 7. Consider, as a first candidate for contract composition, the composition $\mathscr{C}_1 \times \mathscr{C}_2$ where $\mathscr{C}_i, i = 1, 2$ 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

$$E \models^{\mathrm{E}} \mathscr{C} \implies [E \times M_2 \models^{\mathrm{E}} \mathscr{C}_1 \text{ and } E \times M_1 \models^{\mathrm{E}} \mathscr{C}_2]$$

which requires: $\forall \alpha \in \Sigma_i^{\mathrm{out}}$: $q_i \xrightarrow{\alpha}_i \implies (q_1, q_2) \xrightarrow{\alpha}_{\mathscr{C}_1 \times \mathscr{C}_2}$. 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 does not have the corresponding input hence preventing a synchronization—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 $\mathscr{C}_1 \times \mathscr{C}_2$ together with corresponding incoming transitions may cause other illegal pairs to occur. The latter must also be pruned away, until a fixed-point is reached. The result is the contract composition $\mathscr{C}_1 \otimes \mathscr{C}_2$.

As a consequence of this pruning, it may be the case that the resulting contract has an empty set of states, which, by Lemma 7, expresses that the resulting composition of the two interfaces is inconsistent and incompatible—in the original literature on Interface Automata [84, 6] it is said that the two interfaces $\mathscr{C}_1$ and $\mathscr{C}_2$ are incompatible.

In [42], incremental design of deterministic Interface Automata is studied. Let $\mathscr{C}^{\downarrow}$ be the interface $\mathscr{C}$ with input and output actions interchanged. Given two Interface Automata $\mathscr{C}_1$ and $\mathscr{C}_2$, the greatest interface compatible with $\mathscr{C}_2$ such that their composition refines $\mathscr{C}_1$ is given by $(\mathscr{C}_1 \parallel \mathscr{C}_2{}^{\downarrow})^{\downarrow}$. 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.[28] 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.

5.3 Modal Interfaces with fixed alphabet

Modal Interfaces inherit from both the Interface Automata and the originally unrelated notion of Modal Automaton (or Modal Transition System), see the bibliographical note 5.8. As for Interface Automata, the semantics of Modal Interfaces is given

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

below in terms of contracts, that is, pairs formed of a component and of a valid environment represented as receptive i/o-automata. The presentation of Modal Interfaces we develop here is thus aligned with our meta-theory and, thus, differs from classical presentations. Again, we begin with the case of a fixed alphabet Σ.

Definition 8 *Call* Modal Interface *a tuple* $\mathscr{C} = (\Sigma^{\text{in}}, \Sigma^{\text{out}}, Q, q_0, \rightarrow, \dashrightarrow)$*, where* $\Sigma^{\text{in}}, \Sigma^{\text{out}}, Q, q_0$ *are as in Interface Automata and* $\rightarrow, \dashrightarrow \subseteq Q \times \Sigma \times Q$ *are two deterministic transition relations such that*

$$q \xrightarrow{\alpha} q' \text{ and } q \overset{\alpha}{\dashrightarrow} q'' \text{ implies } q' = q'' \,, \tag{38}$$

called must *and* may*, respectively. A Modal Interface* $\mathscr{C}$ *such that* $q_0 \in Q$ *induces two (possibly non receptive) i/o-automata:*

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

$\mathscr{C}$ defines a contract by fixing a pair $(\mathcal{E}_{\mathscr{C}}, \mathcal{M}_{\mathscr{C}})$ *as follows:*

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

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

2. *For any* $\alpha \in \Sigma^{\text{in}}$ *such that* $q_E \xrightarrow{\alpha}_E$ *and any reachable state* (q_E, q) *of* $E \times \mathscr{C}^{may}$*:* $(q_E, q) \xrightarrow{\alpha}_{E \times \mathscr{C}^{must}}$.

Now define the particular environment $E_{\mathscr{C}} \in \mathcal{E}_{\mathscr{C}}$*, such that* $E_{\mathscr{C}} \times \mathscr{C}^{may}$ *simulates* $E \times \mathscr{C}^{may}$ *in the sense of (34) for any* $E \in \mathcal{E}_{\mathscr{C}}$*; we define* $E_{\mathscr{C}}$ *as follows:*

(a) $E_{\mathscr{C}} = (\Sigma^{\text{out}}, \Sigma^{\text{in}}, Q \cup \{\top\}, q_0, \rightarrow_{E_{\mathscr{C}}})$*, where* $\top$ *is a special extra state;*

(b) the restriction of $\rightarrow_{E_{\mathscr{C}}}$ *to* $Q \times \Sigma^{\text{in}} \times Q$ *coincides with* $\rightarrow$*;*
the restriction of $\rightarrow_{E_{\mathscr{C}}}$ *to* $Q \times \Sigma^{\text{out}} \times Q$ *coincides with* $\dashrightarrow$*;*

(c) we make the resulting i/o-automaton receptive as indicated in (36).

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

3. $E_{\mathscr{C}} \times \mathscr{C}^{may}$ *simulates* $E_{\mathscr{C}} \times M$ *in the sense of (34), meaning that only* may *transitions are allowed for* $E_{\mathscr{C}} \times M$*;*

4. $E_{\mathscr{C}} \times M$ *simulates* $E_{\mathscr{C}} \times \mathscr{C}^{must}$ *in the sense of (34), meaning that* must *transitions are mandatory in* $E_{\mathscr{C}} \times M$*.*

Observe that, since components are receptive i/o-automata, we can equivalently replace $\alpha \in \Sigma_M$ by $\alpha \in \Sigma_M^{\text{out}}$ in the above condition 4. On the other hand, the consideration of the particular environment $E_{\mathscr{C}}$ is justified by the following result, whose proof is similar to that of Lemma 6:

Lemma 8 *The environment* $E_{\mathscr{C}}$ *is maximal in* $\mathcal{E}_{\mathscr{C}}$ *with respect to simulation relation* (34).

Consistency and Compatibility: We begin with consistency. Say that state $q \in Q$ of $\mathscr{C}$ is *consistent* if $q \xrightarrow{\alpha} q'$ implies $q \overset{\alpha}{\dashrightarrow} q'$, otherwise we say that it is *inconsistent*. Assume that $\mathscr{C}$ has some inconsistent state $q \in Q$, meaning that, for some action α, $q \xrightarrow{\alpha} q'$ holds but $q \overset{\alpha}{\dashrightarrow} q'$ does not hold. By conditions 3) and 4) of Definition 8, for any environment E and any implementation M of $\mathscr{C}$, no state (q_E, q_M) of $E \times M$ satisfies $(q_E, q_M) \leq (q_E, q)$. Hence, all may transitions leading to q can be deleted from $\mathscr{C}$ without changing $\mathcal{M}_{\mathscr{C}}$. Performing this makes state q unreachable in $\mathscr{C}^{may}$, thus Condition 2 of Definition 8 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 $Q = Q_{\text{con}} \uplus Q_{\text{incon}}$, 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 $[\mathscr{C}]$ the so obtained contract:

Lemma 9 (reduction) *$[\mathscr{C}]$ is called the* reduction *of $\mathscr{C}$. It satisfies*

$$\mathcal{M}_{[\mathscr{C}]} = \mathcal{M}_{\mathscr{C}} \quad \textit{and} \quad \mathcal{E}_{[\mathscr{C}]} \supseteq \mathcal{E}_{\mathscr{C}} \tag{39}$$

where the inclusion is strict unless $\mathscr{C}$ possesses no inconsistent state. Furthermore, $[\mathscr{C}]$ offers the smallest set of environments among the Modal Interfaces satisfying (39). Finally, $\mathscr{C}$ is consistent and compatible if and only if $q_0 \in Q_{\text{con}}$.

In the sequel, unless otherwise specified, we will only consider reduced Modal Interfaces, whose states are all consistent and compatible.

Introducing must, may, and ready sets: It will be useful for the mathematics to reformulate the conditions of Definition 8 using *must* and *may* sets, and *ready sets* we introduce now. For M a component (i.e., a receptive i/o-automaton) and q a reachable state of it, we denote by

$$\Sigma_M(q) \quad =_{\text{def}} \quad \{\alpha \mid q \xrightarrow{\alpha}_M \} \tag{40}$$

the *ready set* of M at q. For $\mathscr{C}$ a Modal Interface and q a state of it, we introduce the following *may* and *must* sets:

$$\begin{array}{rcl} may_{\mathscr{C}}(q) & = & \{\alpha \in \Sigma \mid q \overset{\alpha}{\dashrightarrow} \} \\ must_{\mathscr{C}}(q) & = & \{\alpha \in \Sigma \mid q \xrightarrow{\alpha} \} \end{array} \tag{41}$$

and we recall that, since $\mathscr{C}$ is assumed reduced, the inclusion $must_{\mathscr{C}}(q) \subseteq may_{\mathscr{C}}(q)$ holds. For both notions, we omit the subscript $_M$ or $_{\mathscr{C}}$ when no confusion can result. Using these notions, the following lemma holds:

Lemma 10

1. *For M_1 and M_2 two i/o-automata as in (33), simulation relation $q_2 \leq q_1$ rewrites as follows: $\Sigma_{M_2}(q_2) \subseteq \Sigma_{M_1}(q_1)$ holds and $\delta_{M_2}(q_2, \alpha) \leq \delta_{M_1}(q_1, \alpha)$ holds for every $\alpha \in \Sigma_{M_2}(q_2)$.*
2. *The conditions of Definition 8 can be reformulated as follows:*

Condition 2. rewrites as follows: *For every pair (q_E, q) of states that is reachable in $E \times \mathscr{C}^{may}$, the following holds:*

$$\Sigma^{\text{in}} \cap \Sigma_E(q_E) \quad \subseteq \quad must_{\mathscr{C}}(q) \tag{42}$$

Condition 3. rewrites as follows: *For every pair (q, q_M) of states that is reachable in $E_{\mathscr{C}} \times M$, the following holds:*

$$\Sigma^{\text{out}} \cap \Sigma_M(q_M) \quad \subseteq \quad \Sigma^{\text{out}} \cap may(q) \tag{43}$$

Condition 4. rewrites as follows: *For every pair (q, q_M) of states that is reachable in $E_{\mathscr{C}} \times M$, the following holds:*

$$\Sigma^{\text{out}} \cap must(q) \quad \subseteq \quad \Sigma^{\text{out}} \cap \Sigma_M(q_M) \tag{44}$$

Proof: Statement 1 and the item related to Condition 2 of Statement 2 are immediate. Condition 3 of Statement 2 is equivalent to the following inclusion, which by itself implies that the special state $\top$ of $E_{\mathscr{C}}$ is not reachable in the product $E_{\mathscr{C}} \times M$:

$$\left((\Sigma^{\text{in}} \cap must(q)) \cup \Sigma^{\text{out}}\right) \cap \Sigma_M(q_M) \quad \subseteq \quad (\Sigma^{\text{in}} \cap must(q)) \cup (\Sigma^{\text{out}} \cap may(q)) \tag{45}$$

Using the partitioning $\Sigma = \Sigma^{\text{in}} \cup \Sigma^{\text{out}}$, inclusion (45) is equivalent to the conjunction of the following two inclusions:

$$\begin{aligned} \Sigma^{\text{in}} \cap must(q) \cap \Sigma_M(q_M) \quad &\subseteq \quad \Sigma^{\text{in}} \cap must(q) \\ \Sigma^{\text{out}} \cap \Sigma_M(q_M) \quad &\subseteq \quad \Sigma^{\text{out}} \cap may(q) \end{aligned}$$

which is equivalent to (43) since the first inclusion is a tautology. The reasoning for Condition 4 is similar. First, Condition 4 is equivalent to

$$must(q) \quad \subseteq \quad \left((\Sigma^{\text{out}} \cap may(q)) \cup \Sigma^{\text{in}}\right) \cap \Sigma_M(q_M)$$

which, by decomposing over $\Sigma = \Sigma^{\text{in}} \cup \Sigma^{\text{out}}$, and using (43), proves (44). □ *Relation of Definition 8 with the existing semantics of Modal Interfaces*: The first sentence of Definition 8 is a verbatim of the original definition of Modal Interfaces [183]. As for Interface Automata in Section 5.2, the handling of the initial state q_0 is heterodox and motivated by our aim that Definition 8 casts Modal Interfaces in the framework of the meta-theory. For the same reason, Definition 8 introduces the two sets $\mathcal{E}_{\mathscr{C}}$ and $\mathcal{M}_{\mathscr{C}}$, whereas the classical theory of Modal Interfaces considers and develops a different notion of *model* (often also called "implementation", which is unfortunate). Nevertheless, the following relation holds between $\mathcal{M}_{\mathscr{C}}$ and the set of models of $\mathscr{C}$.

Lemma 11

1. *The map $M \to \overline{M}$ defined in (36) maps every model of $\mathscr{C}$ to an implementation of $\mathscr{C}$, i.e., $\overline{M} \in \mathcal{M}_{\mathscr{C}}$.*
2. *For every $M \in \mathcal{M}_{\mathscr{C}}$, $N = \left(\Sigma^{\text{in}}, \Sigma^{\text{out}}, Q_M \times Q, (q_{M,0}, q_0), \to_{M \times E_{\mathscr{C}}}\right)$ is a model of $\mathscr{C}$.*

Proof: For statement 1, we must prove that every model $M = (\Sigma^{\text{in}}, \Sigma^{\text{out}}, Q_M, q_0^M, \to_M)$ of $\mathscr{C}$ yields an implementation $\overline{M}$ of $\mathscr{C}$ according to Definition 8. Recall that *M models*

$\mathscr{C}$, written $M \models \mathscr{C}$ if, for every pair of states (q_M, q) that is reachable in $M \times \mathscr{C}^{may}$, the following holds [183]:

$$must(q) \subseteq \Sigma_M(q_M) \quad \subseteq \quad may(q) \tag{46}$$

Using Lemma 10, this condition implies that Conditions 3 and 4 of Definition 8 hold for M. Replacing M by its receptive counterpart $\overline{M}$ does not change anything since the transitions added when moving from M to $\overline{M}$ are canceled in the composition $E_{\mathscr{C}} \times M$.

For statement 2, we need to prove that, for every triple (q_M, q, q) of states that is reachable in $N \times \mathscr{C}^{may}$,

$$must(q) \subseteq \Sigma_N(q_M, q) \quad \subseteq \quad may(q) \tag{47}$$

By construction, we have

$$\Sigma_N(q_M, q) \quad = \quad \Sigma_M(q_M) \cap \left((\Sigma^{\text{in}} \cap must(q)) \cup (\Sigma^{\text{out}} \cap may(q))\right) \tag{48}$$

Using again Lemma 10, Conditions 3 and 4 of Definition 8 for M, imply: for every pair (q, q_M) of states that is reachable in $E_{\mathscr{C}} \times M$,

$$\Sigma^{\text{out}} \cap must(q) \quad \subseteq \quad \Sigma^{\text{out}} \cap \Sigma_M(q_M) \quad \subseteq \quad \Sigma^{\text{out}} \cap may(q) \tag{49}$$

Intersecting the second inclusion of (49) with $may(q)$ and using (48) yields

$$\Sigma^{\text{out}} \cap \Sigma_N(q_M, q) \quad \subseteq \quad \Sigma^{\text{out}} \cap may(q) \tag{50}$$

On the other hand, intersecting the first inclusion of (49) with $must(q)$ and using (48) and the fact that $must(q) \subseteq may(q)$ gives

$$\Sigma^{\text{out}} \cap must(q) \quad \subseteq \quad \Sigma^{\text{out}} \cap \Sigma_N(q_M, q) \tag{51}$$

Finally, using once more (48) and the fact that M is receptive, we get $\Sigma^{\text{in}} \cap \Sigma_N(q_M, q) = \Sigma^{\text{in}} \cap must(q)$, which, together with (50) and (51), yields (47). □

Refinement and conjunction: Conjunction and refinement are instantiated in a very elegant way in the theory of Modal Interfaces. Contract refinement in the sense of the meta-theory is instantiated by the effective notion of Modal refinement we introduce now. Roughly speaking, modal refinement consists in enlarging the must relation (thus enlarging the set of legal environments) and restricting the may relation (thus restricting the set of implementations). The formalization requires the use of simulation relations.

Definition 9 (modal refinement) *Let $\mathscr{C}_i, i = 1, 2$ be two Modal Interfaces and q_i be a state of $\mathscr{C}_i$, for $i = 1, 2$. Say that q_2* refines *q_1, written $q_2 \preceq q_1$, if:*

$$\left\{ \begin{array}{lcl} may_2(q_2) & \subseteq & may_1(q_1) \\ must_2(q_2) & \supseteq & must_1(q_1) \end{array} \right. \quad \text{and} \quad \forall \alpha \in \Sigma : \left\{ \begin{array}{l} q_1 \overset{\alpha}{\dashrightarrow}_1 q_1' \\ q_2 \overset{\alpha}{\dashrightarrow}_2 q_2' \end{array} \right. \implies q_2' \preceq q_1'$$

Say that $\mathscr{C}_2 \preceq \mathscr{C}_1$ 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 8 and Lemma 9:

Lemma 12 *For $\mathscr{C}_1$ and $\mathscr{C}_2$ two Modal Interfaces, the following two properties are equivalent:*

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

(ii) $[\mathscr{C}_2] \preceq [\mathscr{C}_1]$*, i.e., $[\mathscr{C}_2]$ refines contract $[\mathscr{C}_1]$ 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 10 *The* pre-conjunction[29] *$\mathscr{C}_1 \underline{\wedge} \mathscr{C}_2$ of two Modal Interfaces is only defined if $\Sigma_1^{\text{in}} = \Sigma_2^{\text{in}}$ and $\Sigma_1^{\text{out}} = \Sigma_2^{\text{out}}$ and is given by $\Sigma^{\text{in}} = \Sigma_1^{\text{in}}$, $\Sigma^{\text{out}} = \Sigma_1^{\text{out}}$, $Q = (Q_1 \times Q_2) \cup \{\bot\}$, $q_0 = (q_{1,0}, q_{2,0})$, and its two transition relations are given by:*

$$\begin{array}{rcl}
(q_1,q_2) \xrightarrow{\alpha} (q_1',q_2') & \text{iff} & q_i \xrightarrow{\alpha}_i q_i', \text{ for } i = 1,2 \\
(q_1,q_2) \xrightarrow{\alpha} \bot & \text{iff} & q_i \xrightarrow{\alpha}_i q_i' \text{ and } \alpha \notin \mathit{must}_j(q_j), \text{ for } i,j = 1,2, j{\neq}i \\
(q_1,q_2) \xrightarrow{\alpha}\!\!\!\!\!\rightarrow (q_1',q_2') & \text{iff} & q_i \xrightarrow{\alpha}\!\!\!\!\!\rightarrow_i q_i', \text{ for } i = 1,2
\end{array}$$

By construction, the *must* and *may* sets of $\mathscr{C}_1 \underline{\wedge} \mathscr{C}_2$ are given by:

$$\begin{array}{rcl}
\mathit{must}(q_1,q_2) & = & \mathit{must}_1(q_1) \cup \mathit{must}_2(q_2) \\
\mathit{may}(q_1,q_2) & = & \mathit{may}_1(q_1) \cap \mathit{may}_2(q_2)
\end{array} \tag{52}$$

Now by (52), we can see that $\mathscr{C}_1 \underline{\wedge} \mathscr{C}_2$ may involve inconsistent states and, thus, in a second step, the pruning introduced in Lemma 9 must be applied:

$$\mathscr{C}_1 \wedge \mathscr{C}_2 \quad = \quad [\mathscr{C}_1 \underline{\wedge} \mathscr{C}_2] \tag{53}$$

Say that *the two Modal Interfaces $\mathscr{C}_1$ and $\mathscr{C}_2$ are consistent* if $\mathscr{C}_1 \wedge \mathscr{C}_2$ is consistent in the sense of Lemma 9.

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, thus, a two-step procedure is again followed for its computation.

Definition 11 *The* pre-composition *$\mathscr{C}_1 \underline{\otimes} \mathscr{C}_2$ of two Modal Interfaces is only defined if $\Sigma_1^{\text{out}} \cap \Sigma_2^{\text{out}} = \emptyset$ and is given by: $\Sigma^{\text{out}} = \Sigma_1^{\text{out}} \cup \Sigma_2^{\text{out}}$, $Q = Q_1 \times Q_2$, $q_0 = (q_{1,0}, q_{2,0})$, and its two transition relations are given by:*

$$\begin{array}{rcl}
(q_1,q_2) \xrightarrow{\alpha} (q_1',q_2') & \text{iff} & q_i \xrightarrow{\alpha}_i q_i', i = 1,2 \\
(q_1,q_2) \xrightarrow{\alpha}\!\!\!\!\!\rightarrow (q_1',q_2') & \text{iff} & q_i \xrightarrow{\alpha}\!\!\!\!\!\rightarrow_i q_i', i = 1,2
\end{array}$$

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

$$\begin{array}{rcl}
\mathit{may}_1(q_1) \cap \Sigma_2^{\text{in}} & \not\subseteq & \mathit{must}_2(q_2) \\
\text{or } \mathit{may}_2(q_2) \cap \Sigma_1^{\text{in}} & \not\subseteq & \mathit{must}_1(q_1)
\end{array}$$

[29] Pre-conjunction was originally denoted by the symbol & in [181, 184, 183].

Illegal states are pruned away from $\mathscr{C}_1 \underline{\otimes} \mathscr{C}_2$ as follows. Remove, from $\mathscr{C}_1^{may} \times \mathscr{C}_2^{may}$, 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 $\mathscr{C}_1 \otimes \mathscr{C}_2$, which no more possesses illegal states.

By construction, the *must* and *may* sets of $\mathscr{C}_1 \underline{\otimes} \mathscr{C}_2$ are given by:

$$\begin{array}{rcl} must(q_1, q_2) & = & must_1(q_1) \cap must_2(q_2) \\ may(q_1, q_2) & = & may_1(q_1) \cap may_2(q_2) \end{array} \tag{54}$$

The above construction is justified by the following result:

Lemma 13 *$\mathscr{C}_1 \otimes \mathscr{C}_2$ as defined in Definition 11 instantiates the contract composition from the meta-theory.*

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

$$rs(q_{\underline{E}}) \quad \subseteq \quad must_1(q_1) \cap must_2(q_2), \tag{55}$$

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 $\mathscr{C}_1$ and consider $\underline{E} \times M_1$. We need to prove that $\underline{E} \times M_1$ is an environment for $\mathscr{C}_2$, i.e., satisfies Condition 2 of Definition 8 (we must also prove the symmetric property). Let $(q_{\underline{E}}, q_1, q_2)$ be a reachable state of $\underline{E} \times M_1 \times \mathscr{C}_2^{may}$. We must prove

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

By (55) it suffices that the following property holds:

$$rs_{M_1}(q_1) \cap \Sigma_2^{\text{in}} \quad \subseteq \quad must_2(q_2) \tag{56}$$

However, we only know that $rs_{M_1}(q_1) \subseteq may_1(q_1)$. Hence, to guarantee (56) 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 $\mathscr{C}_1^{may} \times \mathscr{C}_2^{may}$ by removing all *may* transitions leading to that state. We complete the reasoning as we did for the study of consistency. □

Definition 12 *The* quotient *$\mathscr{C}_1 / \mathscr{C}_2$ of two Modal Interfaces $\mathscr{C}_1$ and $\mathscr{C}_2$ 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 $\mathscr{C}$ as follows: $\Sigma^{\text{out}} = \Sigma_1^{\text{out}} \setminus \Sigma_2^{\text{out}}$, $Q = (Q_1 \times Q_2) \cup \{\bot, \top\}$, $q_0 = (q_{1,0}, q_{2,0})$, and its two transition relations are given by:*

$$\begin{array}{rcll} (q_1, q_2) \xrightarrow{\alpha} (q_1', q_2') & \text{iff} & q_i \xrightarrow{\alpha}_i q_i', i = 1, 2 \\ (q_1, q_2) \xrightarrow{\alpha} \bot & \text{iff} & q_1 \xrightarrow{\alpha}_1 q_1' \text{ and } \alpha \notin must_2(q_2) \\ (q_1, q_2) \dashrightarrow^{\alpha} (q_1', q_2') & \text{iff} & q_1 \dashrightarrow^{\alpha}_1 q_1', q_2 \dashrightarrow^{\alpha}_2 q_2' \text{ and } \alpha \in must_2(q_2) \\ (q_1, q_2) \dashrightarrow^{\alpha} (q_1', q_2') & \text{iff} & q_1 \dashrightarrow^{\alpha}_1 q_1', q_2 \dashrightarrow^{\alpha}_2 q_2' \text{ and } \alpha \notin must_1(q_1) \\ (q_1, q_2) \dashrightarrow^{\alpha} \top & \text{iff} & q_1 \dashrightarrow^{\alpha}_1 q_1' \text{ and } \alpha \in may_2(q_2) \\ (q_1, q_2) \dashrightarrow^{\alpha} \top & \text{iff} & \neg \left[may_1(q_1) \cup may_2(q_2) \right] \\ \top \dashrightarrow^{\alpha} \top & \text{iff} & \alpha \in \Sigma \end{array}$$

These rules may result in inconsistent states, thus, in a second step, we set $\mathscr{C}_1 / \mathscr{C}_2 = [\mathscr{C}]$.

Observe that, by construction, the *must* and *may* sets of $\mathscr{C}_1/\mathscr{C}_2$ are given by:

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

This definition is justified by the following result [183], showing that Definition 12 instantiates the meta-theory:

Lemma 14 *$\mathscr{C} \otimes \mathscr{C}_2 \preceq \mathscr{C}_1$ if and only if $\mathscr{C} \preceq \mathscr{C}_1/\mathscr{C}_2$.*

For the next definition and lemma, we consider two consistent and compatible Modal Interfaces $\mathscr{C}_1$ and $\mathscr{C}_2$. We are interested in modifying the quotient $\mathscr{C}_1/\mathscr{C}_2$ to prevent $(\mathscr{C}_1/\mathscr{C}_2)$ and $\mathscr{C}_2$ from being incompatible. For the following definition, Σ^{out} and Σ^{in} are as in Definition 12 and $may_/$ and $must_/$ refer to the quotient $\mathscr{C}_1/\mathscr{C}_2$:

Definition 13 *Let $\mathscr{C}'$ be the interface defined on the same state structure as $\mathscr{C}_1/\mathscr{C}_2$, with however the following modalities:*

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

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

This construction is justified by the following result:

Lemma 15 *The compatible quotient satisfies:*

$$\mathscr{C}_1 /\!\!/ \mathscr{C}_2 = \max \left\{ \mathscr{C} \;\middle|\; \begin{array}{l} \mathscr{C} \text{ has no inconsistent state} \\ \mathscr{C} \otimes \mathscr{C}_2 \text{ has no illegal pair of states} \\ \mathscr{C} \otimes \mathscr{C}_2 \preceq \mathscr{C}_1 \end{array} \right\}$$

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

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

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

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

- Assume $e \in \Sigma_1^{\text{in}}$, meaning that e is an input for both $\mathscr{C}_2$ and $\mathscr{C}_1 /\!\!/ \mathscr{C}_2$. 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 $\mathscr{C}''$ be a Modal Interface such that $\mathscr{C}_1 /\!\!/ \mathscr{C}_2 \preceq \mathscr{C}'' \preceq \mathscr{C}_1 / \mathscr{C}_2$. We shall prove that either $\mathscr{C}'' \preceq \mathscr{C}_1 /\!\!/ \mathscr{C}_2$ or that $\mathscr{C}''$ is not compatible with $\mathscr{C}_2$. Remark that every reachable state (q_1, q_2) of $\mathscr{C}_1 /\!\!/ \mathscr{C}_2$ is related to exactly one state q'' of $\mathscr{C}''$ by the two modal refinement relations. Assume that $\mathscr{C}''$ is not a refinement of $\mathscr{C}_1 /\!\!/ \mathscr{C}_2$, 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 $\mathscr{C}'' \otimes \mathscr{C}_2$.

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 $\mathscr{C}'' \otimes \mathscr{C}_2$. □

5.4 Modal Interfaces with variable alphabet

As a general principle, every relation or operator introduced in Section 5.3 (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 5.3 to the resulting Modal Interfaces. For all frameworks we studied so far, alphabet extension was performed using inverse projections, see Section 4.3. 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 (52) or (54) yields two different answers, namely:

$$\text{for (52)} \quad : \quad \left.\begin{array}{ll} & \alpha \in may_1(q_1) \\ \text{and} & \alpha \in whatever_2(q_2) \end{array}\right\} \implies \alpha \in whatever(q_1, q_2)$$

$$\text{for (54)} \quad : \quad \left.\begin{array}{ll} & \alpha \in must_1(q_1) \\ \text{and} & \alpha \in whatever_2(q_2) \end{array}\right\} \implies \alpha \in whatever(q_1, q_2)$$

where "*whatever*" denotes either *may* or *must*. Consequently, neutral alphabet extension is by adding

- *may* self-loops for the conjunction, and
- *must* self-loops for the composition.

The bottom line is that we need *different extension procedures.* These observations explain why alphabet extension is properly handled neither by Interface Automata (see the last paragraph of Section 5.2) nor by A/G contracts (see the end of Section 4.3).

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 $\mathscr{C}_1$ and $\mathscr{C}_2$ (the reader is referred to [183] for justifications).

Throughout this section, and, more generally, when alphabet extensions are considered, every alphabet comes with its partitioning $\Sigma = \Sigma^{\text{in}} \uplus \Sigma^{\text{out}}$ and $\Sigma' \supseteq \Sigma$ means $\Sigma'^{\text{in}} \supseteq \Sigma^{\text{in}}$ and $\Sigma'^{\text{out}} \supseteq \Sigma^{\text{out}}$.

With this in mind, we define the *strong extension* of $\mathscr{C}$ to $\Sigma' \supseteq \Sigma$, written $\mathscr{C}^{\uparrow\Sigma'}$, as the modal interface $\mathscr{C}^{\uparrow\Sigma'} = (\Sigma'^{\text{in}}, \Sigma'^{\text{out}}, Q, q_0, \rightarrow', \dashrightarrow')$, where:

$$\begin{array}{rcl} \rightarrow' & = & \rightarrow \cup \{\, (q, \alpha, q) \mid q \in Q \text{ and } \alpha \in \Sigma' \setminus \Sigma \,\} \\ \dashrightarrow' & = & \dashrightarrow \cup \{\, (q, \alpha, q) \mid q \in Q \text{ and } \alpha \in \Sigma' \setminus \Sigma \,\} \end{array} \tag{57}$$

Similarly, we define the *weak extension* of $\mathscr{C}$ to $\Sigma' \supseteq \Sigma$, written $\mathscr{C}^{\Uparrow\Sigma'}$, as the modal interface

$$\mathscr{C}^{\Uparrow\Sigma'} = (\Sigma'^{\text{in}}, \Sigma'^{\text{out}}, Q, q_0, \rightarrow, \dashrightarrow') \tag{58}$$

where $\dashrightarrow'$ is defined as in (57) while $\rightarrow$ is kept unchanged. In words, only *may* self-loops are added in weak extensions, whereas both *may* and *must* self-loops are added in the strong extension.

Observe that the strong extension uses the classical inverse projection everywhere. The weak extension, however, proceeds differently with the *must* transitions in that it forbids the legal environments to submit additional actions as its outputs.

Using weak and strong alphabet equalization, the relations and operations introduced in Section 5.3 extend to variable alphabets as indicated now. In the following theorem, (Σ, Σ') is a pair such that $\Sigma \subseteq \Sigma'$ and $(\Sigma_1, \Sigma_2, \Sigma)$ denotes a triple such that $\Sigma = \Sigma_1 \cup \Sigma_2$. Contract $\mathscr{C}$ has alphabet Σ, and contract $\mathscr{C}_i$ has alphabet Σ_i. Finally, for each listed operation, decomposition $\Sigma_i = \Sigma_i^{\text{in}} \uplus \Sigma_i^{\text{out}}$ is such that composability conditions are satisfied:

Theorem 4 *The following relations and operators*

$$\begin{array}{rcl} M' \models^{\text{M}} \mathscr{C} & ::= & M' \models^{\text{M}} \mathscr{C}^{\Uparrow\Sigma'} \\ E' \models^{\text{E}} \mathscr{C} & ::= & E' \models^{\text{E}} \mathscr{C}^{\uparrow\Sigma'} \\ \mathscr{C}_1 \preceq \mathscr{C}_2 & ::= & \mathscr{C}_1 \preceq \mathscr{C}_2^{\Uparrow\Sigma} \\ \mathscr{C}_1 \wedge \mathscr{C}_2 & ::= & \mathscr{C}_1^{\Uparrow\Sigma} \wedge \mathscr{C}_2^{\Uparrow\Sigma} \\ \mathscr{C}_1 \otimes \mathscr{C}_2 & ::= & \mathscr{C}_1^{\uparrow\Sigma} \otimes \mathscr{C}_2^{\uparrow\Sigma} \\ \mathscr{C}_1 /\!\!/ \mathscr{C}_2 & ::= & \mathscr{C}_1^{\Uparrow\Sigma} /\!\!/ \mathscr{C}_2^{\uparrow\Sigma} \end{array} \tag{59}$$

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. □

5.5 Restricting to a sub-alphabet, application to contract decomposition

A difficult step in the management of contracts was illustrated in Figure 2 of Section 2.1. It consists in decomposing a contract $\mathscr{C}$ into a composition of sub-contracts

$$\bigotimes_{i\in I} \mathscr{C}_i \quad \preceq \quad \mathscr{C} \tag{60}$$

where sub-contract $\mathscr{C}_i$ has alphabet $\Sigma_i = \Sigma_i^{\text{in}} \uplus \Sigma_i^{\text{out}}$. As a prerequisite to (60), the designer has to guess some topological architecture by decomposing the alphabet of actions of $\mathscr{C}$ as

$$\Sigma \quad = \quad \bigcup_{i\in I} \Sigma_i \quad , \quad \Sigma_i \quad = \quad \Sigma_i^{\text{in}} \uplus \Sigma_i^{\text{out}} \tag{61}$$

such that composability conditions regarding inputs and outputs hold. Guessing architectural decomposition (61) relies on the designer's understanding of the system and how it should naturally decompose—this typically is the world of SysML. Finding decomposition (60) is, however, technically difficult in that it involves behaviors [143]. It is particularly difficult if $\mathscr{C}$ is itself a conjunction of viewpoints or requirements, which typically occurs in requirements engineering, see companion paper [30]:

$$\mathscr{C} \quad = \quad \bigwedge_{k\in K} \mathscr{C}_k \tag{62}$$

The algorithmic means we develop in the remaining part of this section will be instrumental in solving (60). They will be used in the Parking Garage example of companion paper [30].

Let $\mathscr{C}$ be a Modal Interface with alphabet $\Sigma = \Sigma^{\text{in}} \cup \Sigma^{\text{out}}$ and let $(\Sigma'^{\text{in}}, \Sigma'^{\text{out}})$ be two input and output sub-alphabets such that $\Sigma'^{\text{in}} \subseteq \Sigma^{\text{in}}$ and $\Sigma'^{\text{out}} \subseteq \Sigma^{\text{out}}$. Set $\Sigma' = \Sigma'^{\text{in}} \uplus \Sigma'^{\text{out}}$ and define the *restriction* of $\mathscr{C}$ to Σ', denoted by $\mathscr{C}_{\downarrow\Sigma'}$ via the procedure shown in see Table 4. 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 16 *If $\mathscr{C}$ is consistent, then so are $\mathscr{C}_{\downarrow\Sigma'}$ and the compatible quotient $\mathscr{C} /\!\!/ \mathscr{C}_{\downarrow\Sigma'}$.*

(See Definition 13 for the compatible quotient.) *Proof:* Consider two alphabets $\Sigma \supseteq \Sigma'$, and a consistent $\mathscr{C}$ on alphabet Σ, such that $\mathscr{C}_{\downarrow\Sigma'}$ 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 $\mathscr{C} /\!\!/ \mathscr{C}_{\downarrow\Sigma'}$, 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.

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

let order$(\{cannot, may, must\}, \leq^{\text{in}}) = cannot \leq^{\text{in}} may \leq^{\text{in}} must$

order$(\{cannot, may, must\}, \leq^{\text{out}}) = \begin{cases} must \leq may \\ cannot \leq may \end{cases}$

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 $m = Op\{m_{\mathscr{C}}(q, \alpha) \mid q \in X\}$
     where Op = if $\alpha \in \Sigma'^{\text{in}}$ then $\vee^{\text{in}}$ else $\vee^{\text{out}}$
   - 2.3 add to $\mathscr{C}'$ a transition (X, α, Y) with modality m
   - 2.4 rest(Op, Y)

done

let restrict$(\mathscr{C})$ =

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

Table 4: Algorithm for computing the restriction $\mathscr{C}_{\downarrow\Sigma'}$.

This finishes the proof of the lemma. □ The following properties hold by Lemma 15:

$$\begin{array}{rl} & \mathscr{C}_{\downarrow\Sigma'} \otimes (\mathscr{C} /\!\!/ \mathscr{C}_{\downarrow\Sigma'}) \preceq \mathscr{C}; \\ \mathscr{C} /\!\!/ \mathscr{C}_{\downarrow\Sigma'} & \text{has no inconsistent state;} \\ (\mathscr{C}_{\downarrow\Sigma'}, \mathscr{C} /\!\!/ \mathscr{C}_{\downarrow\Sigma'}) & \text{has no incompatible pair of states.} \end{array} \tag{63}$$

Decomposition (63) can be used while sub-contracting through the following algorithm:

Algorithm 1 *We are given some system-level contract $\mathscr{C}$. The top-level designer guesses some topological architecture according to (61). Then, she recursively decomposes:*

$$\begin{array}{lllllll} \mathscr{C} = \mathscr{C}_0 & \geq & \mathscr{C}_{0\downarrow\Sigma_1} & \otimes & \mathscr{C}_1 & & \\ & \geq & \mathscr{C}_{0\downarrow\Sigma_1} & \otimes & \mathscr{C}_{1\downarrow\Sigma_2} & \otimes & \mathscr{C}_2 \\ & \vdots & & & & & \\ & \geq & \mathscr{C}_{0\downarrow\Sigma_1} & \otimes & \ldots & \otimes & \mathscr{C}_{n-1\downarrow\Sigma_n} \\ & =_{\text{def}} & \mathscr{C}(\Sigma_1) & \otimes & \ldots & \otimes & \mathscr{C}(\Sigma_n) \end{array} \tag{64}$$

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

5.6 Observers

Here we develop observers for a Modal Interface $\mathscr{C} = (\Sigma^{\text{in}}, \Sigma^{\text{out}}, Q, q_0, \rightarrow, \dashrightarrow)$ 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 8 boils down to requiring that $E \times \mathscr{C}^{must}$ simulates E. Simulation testing can thus be used to check this; call $b^{\mathsf{E}}_{\mathscr{C}}(E)$ the corresponding verdict.
- To test for implementations, we first construct the maximal environment $E_{\mathscr{C}}$ and apply testing to check simulation of $E_{\mathscr{C}} \times M$ by $\mathscr{C}^{may}$, call $b^{\mathsf{M}}_{\mathscr{C},1}$ 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 $b^{\mathsf{M}}_{\mathscr{C},2}$ denote the verdict answering whether $(q_E, q_M) \xrightarrow{\alpha}_{E_{\mathscr{C}} \times M}$ holds each time $q \xrightarrow{\alpha}$ holds, for any $\alpha \in \Sigma^{\text{out}}$. The overall verdict for implementation testing is then

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

Lemma 2 for generic observers specializes to the following, effective, semi-decision procedure:

Lemma 17

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

5.7 Using Modal Interfaces to support Assume/Guarantee Contracts

In this section we explain how to represent, using Modal Interfaces, Assume/Guarantee contracts of the form $\mathscr{C} = (\{A_i, \ldots A_n\}, G)$, where the assumptions A_i and the guarantee G are Modal Interfaces. Regarding the i/o status of the assumptions and the guarantee, the following holds:

- The guarantee G specifies the expected behavior of a component. We assume that its i/o alphabet is $\Sigma_G = \Sigma^{\text{in}}_G \uplus \Sigma^{\text{out}}_G$.
- Assumptions A_i adopt the conjugate point of view, since they specify expected properties of the environment. Hence, for $i = 1 \ldots n$, the i/o alphabet $\Sigma_{A_i} = \Sigma^{\text{in}}_{A_i} \uplus \Sigma^{\text{out}}_{A_i}$ of assumption A_i, should be such that:

$$\begin{array}{lcl} \Sigma^{\text{in}}_{A_i} \cap \Sigma^{\text{out}}_{A_j} = \emptyset & \text{for all} & j = 1 \ldots n \\ \Sigma^{\text{in}}_{A_i} \cap \Sigma^{\text{in}}_{G} = \emptyset & \text{and} & \Sigma^{\text{out}}_{A_i} \cap \Sigma^{\text{out}}_{G} = \emptyset \end{array}$$

The next question is: How several assumptions shall be combined together? How do guarantees and assumptions interact?

5.7.1 A vending machine example

These questions are first answered in the context of a simple example: a vending machine serving tea or coffee. This is an academic example distributed as part of the MICA tool [55], an implementation of the Modal Interface theory, supporting contract-based reasoning. This example relates the design of a system with three input actions `?coin`, `?tea_req`, and `?coffee_req`, and two output actions `!tea` and `!coffee`. Question and exclamation marks are only a reminder of the i/o status of the action:

Figure 4: On the left, assumption A_1, "users shall not insert more than one coin per transaction". On the right, assumption A_2, "users shall not press on more than one button per transaction". In both modal specification the initial state is labeled 0 and has a losange shape.

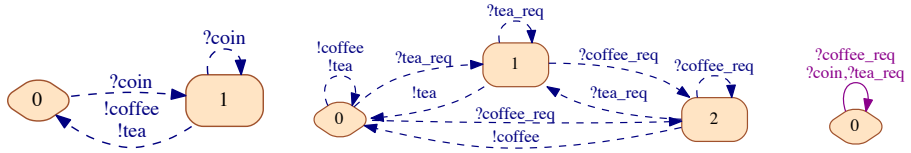


Figure 5: On the left, guarantee G_1, "the machine shall not deliver any beverage before having received payment". In the middle, guarantee G_2, "the machine can not deliver tea when coffee has been requested and deliver coffee when tea has been requested". On the right, guarantee G_3, "the machine must be receptive to input events".

`!` stands for output, and `?` is for input. Assumptions, for this particular example, have no *may* transitions. The reason is twofold: The environment has no control on the output actions of the vending machine, hence `?tea` and `?coffee` transitions have the modality *must*. Regarding the actions under control of the environment, `!coin`, `!tea_req` and `!coffee_req`, the most permissive environment is considered, which explains that these transitions also have a *must* modality.

The behavior of the vending machine is specified as a set of assumptions and guarantees, and a set of contracts relating the previously defined assumptions/guarantees. The assumptions defined in Figures 4 state that the user is expected to insert not more than one coin (assumption A_1) and press more than one button (assumption A_2) per transaction. The expected behavior of the vending machine is also specified in a modular way, with the modal interfaces in Figure 5. These specifications will be used as guarantees, to be paired with assumptions. Guarantee G_1 states that the machine shall not deliver any beverage before having received payment. Guarantee G_2 expresses that the machine can not deliver tea when coffee has been requested and deliver coffee when tea has been requested. Guarantee G_3 simply states that the vending machine must be receptive to its input actions, meaning that it can not refuse `?coin`, `?tea_req`, `?coffee_req`.

Three contracts are considered: $\mathscr{C}_i = (\{A_1, A_2\}, G_i)$, $i = 1 \ldots 3$. Contract $\mathscr{C}_i$ states that G_i must hold, under the assumption that both A_1 and A_2 hold. We capture this by stating that *assumptions compose using the conjunction operator*, to form a global assumption shown in Figure 6, on the top left:

$$A \quad =_{\text{def}} \quad A_1 \wedge A_2.$$

Consider guarantee G_1. Although their alphabets are compatible, specifications G_1 and A have different alphabets, and, thus, the issue of alphabet equalization must be considered. It turns out that alphabet equalization is not performed in the same way for

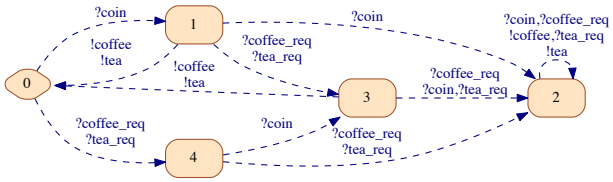


Figure 6: Top left, global guarantee $A = A_1 \wedge A_2$. Top right, $G_1 \ltimes A$, guarantee G_1 put in the context of assumption A. Bottom, contract $\mathscr{C}_1 = (G_1 \ltimes A)/A$

guarantees and assumptions.

- Regarding assumptions:

$$\begin{array}{lcl} \Sigma'^{\,\text{in}}_A & = & \Sigma^{\text{in}}_A \cup \Sigma^{\text{out}}_G \\ \Sigma'^{\,\text{out}}_A & = & \Sigma^{\text{out}}_A \cup \Sigma^{\text{in}}_G \end{array}$$

  Strong equalization is used on assumptions, meaning that self-loop transitions with a *must* modality are inserted in every state of the assumption and for every action in the alphabet of the guarantee that is missing in the alphabet of the assumption. Remark that, for the particular instance of assumption A in our vending machine example, no equalization needs to be performed, and $A' = A$ as a result of equalization.

- Regarding guarantees:

$$\begin{array}{lcl} \Sigma'^{\,\text{in}}_G & = & \Sigma^{\text{in}}_G \cup \Sigma^{\text{out}}_A \\ \Sigma'^{\,\text{out}}_G & = & \Sigma^{\text{out}}_G \cup \Sigma^{\text{in}}_A \end{array}$$

  The same operation applies to the guarantee, using however weak equalization, where *may* self-loops are inserted. The rationale for using *weak equalization on guarantees* is simply that the equalized guarantee G'_1 should be neutral to actions that are not observable by the guarantee.

The equalized guarantee G'_1 and assumption A' are composable and their composition

$$G_1 \ltimes A \quad =_{\text{def}} \quad G'_1 \otimes A' \tag{65}$$

(see Figure 6, top right) is the guarantee G_1 put in the context of assumption A. It specifies the set of compositions of a system satisfying the guarantee G_1, with an environment satisfying assumption A. The contract is then computed by releaving $G_1 \ltimes A$ of the assumption A. This is defined using the quotient operator, shown at the bottom of Figure 6:

$$\mathscr{C}_1 \quad =_{\text{def}} \quad (G_1 \ltimes A)/A \tag{66}$$

Contracts $\mathscr{C}_2 = (G_2 \ltimes A)/A$ and $\mathscr{C}_3 = (G_3 \ltimes A)/A$ are defined in the same way and are shown the top and middle of Figure 7. The global contract is the conjunction of the three contracts $\mathscr{C} =_{\text{def}} \mathscr{C}_1 \wedge \mathscr{C}_2 \wedge \mathscr{C}_3$, shown at the bottom of Figure 7. This contract is clearly incomplete, since it allows implementations of the vending machine that output `!tea` or `!coffee` without any request `?tea_req` or `?coffee_req`. Completing the specification of the vending machine with a fourth contract is an easy exercice, left to the reader.

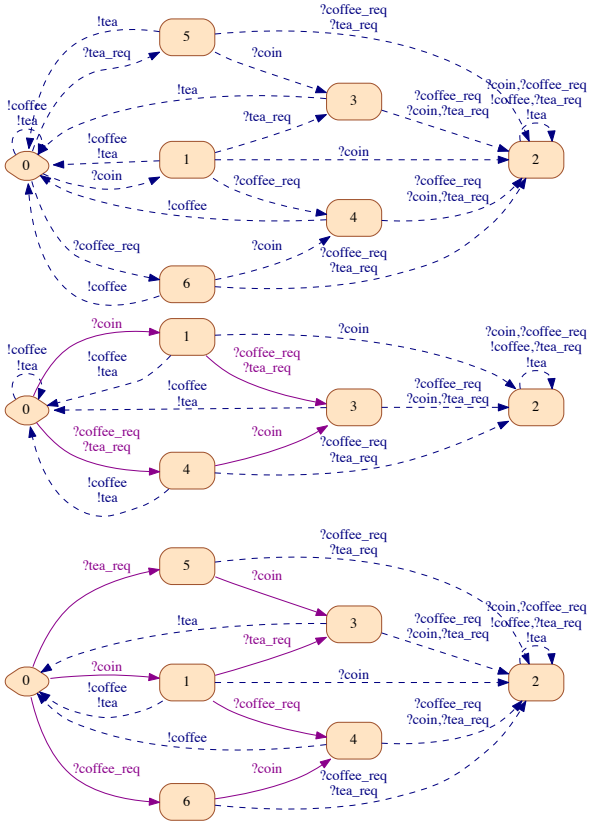


Figure 7: Top, contract $\mathscr{C}_2$. Middle, contract $\mathscr{C}_3$. Bottom, global contract $\mathscr{C} = \mathscr{C}_1 \wedge \mathscr{C}_2 \wedge \mathscr{C}_3$

5.7.2 Comparison with A/G contracts of Section 4

Modal A/G contracts developed in Section 5.7.1 bear many similarities with the A/G contracts detailed in Section 4. In this section we show how A/G contracts can be mapped to Modal A/G contracts in such a way that some, but not all properties of the A/G contract algebra are preserved. Before doing this, we observe the following discrepancies, which prevent a perfect matching:

- The A/G contracts of section 4 are oblivious to event I/O orientation, while in Modal Interfaces, events are either an output or an input, and this plays an important role in the theory.
- A second difference is that A/G contracts are based on a dataflow or synchronous semantics, where behavior is defined as streams of values, one per variable, or as a sequence of partial assignments of the variables. This differs from Modal Interfaces, where behavior is defined as sequences of events taken in a finite alphabet. So far the above two discrepancies can be seen to be technical. The next one, however, is more fundamental.
- In A/G contracts $\mathscr{C} = (A, G)$, the assumption A is handled in a rigid way: $E \models^{\mathrm{E}} \mathscr{C}$ amounts to $E \subseteq A$, and, as a consequence, $\mathscr{C}' \preceq \mathscr{C}$ requires $A' \supseteq A$. In contrast,

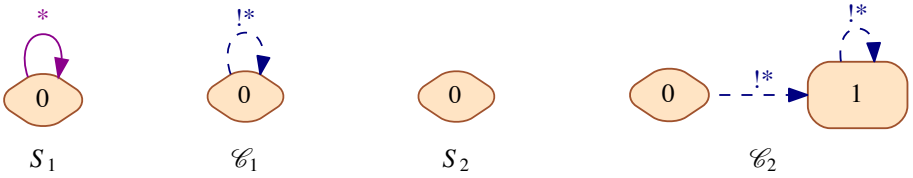


Figure 8: Assumption and guarantee S_1 is shown on the far left. The $*$ symbol stands for any element of alphabet Σ. This is a trivial assertion by which every behavior of the system is permitted. The resulting Modal Interface contract $\mathscr{C}_1 = (S_1 \ltimes S_1)/S_1$ is shown on the center left. It is a trivial interface, satisfied by every transition system on alphabet Σ. Assumption and guarantee S_2 is shown at the center right. Its meaning is that "no event shall happen in the system". The resulting Modal Interface contract $\mathscr{C}_2 = (S_2 \ltimes S_2)/S_2$ si shown at the far right. As a matter of fact, $\mathscr{C}_1$ and $\mathscr{C}_2$ are equivalent, meaning that they refine one-another. Nevertheless, the A/G contracts (S_i, S_i) are incomparable wrt. contract refinement.

formula $\mathscr{C} = (G \ltimes A)/A$ does not define A uniquely and, thus, refinement cannot constrain A directly.

These discrepancies explain why the two theories cannot perfectly match, an one can only hope for a partial embedding of A/G contracts in the Modal Interface theory. This is detailed below. In this development, we use the subscripts $_{\mathsf{AG}}$ and $_{\mathsf{MI}}$ to distinguish contract relations or operations according to the A/G contract and Modal Interface frameworks.

We are given a finite alphabet Σ and we consider the A/G contract (A, G), where A and G are non-empty prefix-closed regular subsets of $\Sigma^\star$.[30] A and G are the languages of deterministic finite transition systems, which we also denote by $A = (\Sigma, Q^A, q_0^A, \rightarrow^A)$ and $G = (\Sigma, Q^G, q_0^G, \rightarrow^G)$. Assumptions and guarantees are mapped to Modal Interfaces:

$$\begin{aligned} A^{\mathsf{m}} &= (\Sigma^{\text{in}} = \emptyset, \Sigma^{\text{out}} = \Sigma, Q^A, q_0^A, \rightarrow^A, \rightarrow^A) \\ G^{\mathsf{m}} &= (\Sigma^{\text{in}} = \emptyset, \Sigma^{\text{out}} = \Sigma, Q^G, q_0^G, \emptyset, \rightarrow^G) \end{aligned}$$

meaning that assumptions are mapped to rigid interfaces, where all transitions are *must* transitions and all actions are outputs, whereas guarantees are mapped to relaxed interfaces with output actions and only *may* transitions. Recall that Modal A/G contracts are given by:

$$\mathscr{C} = (G^{\mathsf{m}} \ltimes A^{\mathsf{m}})/A^{\mathsf{m}}$$

The so defined mapping $(A, G) \rightarrow \mathscr{C}$ preserves refinement:

Theorem 5 *$(A_1, G_1) \preceq_{\mathsf{AG}} (A_2, G_2)$ implies $\mathscr{C}_1 \preceq_{\mathsf{MI}} \mathscr{C}_2$.*

The converse implication does not hold in general, as shown by the counter-example of Figure 8. The reason is that A/G contract refinement requires that assumptions A_1 and A_2 are comparable, while they are not directly related by the Modal Interface refinement relation.

[30] With reference to Section 4 and particularly formula (19), we consider sets of behaviors for a singleton variable v whose domain is $D_v = \Sigma$.

Proof: We use notations from Section 4. Denote by M_i the language of the maximal implementations of $\mathscr{C}_i$. Observe that $G_i \ltimes A_i$ has an empty must transition relation. Therefore, the must transition relation of $\mathscr{C}_i$ is also empty. Therefore $\mathscr{C}_1 \preceq_{\mathsf{MI}} \mathscr{C}_2$ reduces to $M_1 \subseteq M_2$. Denote by $G'_i = (G_i \cup \neg A_i)^{\downarrow}$ the saturated guarantee. Recall that G'_i is the largest prefix-closed language contained in $G_i \cup \neg A_i$. By construction, the language of the *may* transition relation of $G_2^{\mathsf{m}} \ltimes A_i^{\mathsf{m}}$ is $G_i \cap A_i$. Therefore the language of the *may* transition relation of $\mathscr{C}_i$ is equal to $((G_i \cap A_i) \cup \neg A_i)^{\downarrow} = G'_i$. Hence $\mathscr{C}_1 \preceq_{\mathsf{MI}} \mathscr{C}_2$ iff $G'_1 \subseteq G'_2$, which concludes the proof. □ Regarding contract composition, the following theorem states that the Modal Interface image of the composition of two contracts (A_i, G_i), $i = 1, 2$ is equivalent to the composition of the images of the two contracts. Define the images of the three A/G contracts as Modal Interfaces:

$$\begin{aligned} (A_0, G_0) &= (A_1, G_1) \otimes_{\mathsf{AG}} (A_2, G_2), \text{ and} \\ \mathscr{C}_i &= (G_i^{\mathsf{m}} \ltimes A_i^{\mathsf{m}})/A_i^{\mathsf{m}} \quad \text{for } i = 0, 1, 2 \end{aligned}$$

For $\mathscr{C} = (\Sigma^{\text{in}}, \Sigma^{\text{out}}, Q, q_0, \rightarrow, \dashrightarrow)$ a Modal Interface, we define its dual $\overline{\mathscr{C}}$ obtained by exchanging, in $\mathscr{C}$, the input/output status.

Theorem 6 *Modal Interfaces $\mathscr{C}_0$ and $\overline{\mathscr{C}_1} \otimes_{\mathsf{MI}} \mathscr{C}_2$ refine one-another.*

Proof: Modal Interfaces $\mathscr{C}_i$, $i = 0 \ldots 2$ are consistent and their *must* transition relations are empty. Denote by M_i the maximal implementation of $\mathscr{C}_i$. Using the same reasoning as in the previous proof, $\mathscr{C}_0 \equiv \overline{\mathscr{C}_1} \otimes \mathscr{C}_2$ reduces to $M_0 = M_1 \cap M_2$. By definition of contract composition $G_0 = G'_1 \cap G'_2$ and $A_0 = \max\{A \,|\, A = A^{\downarrow}, A \cap G'_2 \subseteq A_1, A \cap G'_1 \subseteq A_2\}$, where $G'_i = (G_i \cup \neg A_i)^{\downarrow}$, $i = 1, 2$ are the saturated guarantees. By construction, $M_i = (G_i \cup \neg A_i)^{\downarrow}$. Using the fact that G_0 is saturated, the definition of M_0 expands to: $M_0 = (G_1 \cup \neg A_1)^{\downarrow} \cap (G_2 \cup \neg A_2)^{\downarrow} = M_1 \cap M_2$.

5.8 Bibliographical note

As explained in [84, 63, 139, 96, 182, 183], Interface Theories make no explicit distinction between assumptions and guarantees. These notions are implicitly supported, however, through the particular semantics of these models.

Interface Automata, variants and extensions: Interface Automata were proposed by de Alfaro and Henzinger [84, 82, 6, 61] as a candidate theory of interfaces. In these references, Interface Automata focused primarily on parallel composition and compatibility. Quoting from [84]: *"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 [84, 63]. In contrast to our development in Section 5.2, no sets of environments and implementations are formally associated to an Interface Automaton in the original developments of the concept. A refinement relation for Interface Automata was defined in [84]—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 [10], 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). Finally, the recent work [51] revisits the foundations of Interface Automata.

In [96], the framework of *Synchronous Interfaces* was enriched with a notion of conjunction (called *shared refinement*). This development was further elaborated in [90] for the topic of time-triggered scheduling. *Synchronous Relational Interfaces* [199, 200] 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 $\mathscr{C}_1$ and $\mathscr{C}_2$, the first one consists in connecting some of the output variables of $\mathscr{C}_1$ to some of the input variables of $\mathscr{C}_2$ 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 [124] studies conditions that need to be imposed on interface models in order to enforce independent implementability with respect to conjunction.

An algebraic theory of interface automata was recently proposed in [67] following a line similar to [68]. Specifications (called "components" in that reference) are characterized by two sets of observable and inconsistent prefix-closed abstract sets of traces constituting the *interface* of the specification. Refinement, conjunction and disjunction, parallel composition, and quotient, are provided, thus offering a comprehensive framework.

Sociable Interfaces [83] combine the approach presented in the previous paragraph with interface automata [84, 85] by enabling communication via shared variables and actions[31]. 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 [5]. *Software Interfaces* were proposed in [62], as a pushdown extension of interface automata (which are finite state). Pushdown interfaces are needed to model call-return stacks of possibly recursive software components. This paper contains also a comprehensive interface description of Tiny OS,[32] 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* [63]. In Moore Interfaces, each variable is either an input or an output, and this status does not change in time. Bidirectional Interfaces offer added flexibility by allowing variables to change I/O status, depending on the local state of the interface. Communication by shared variable is thus supported and, for instance, allows distributed protocols or shared buses to be modeled. In both formalisms, two interfaces are deemed compatible whenever no variable is an output of both interfaces at the same time, and every legal valuation of the output variables of one interface satisfies the input predicate of the other. The main result of the paper is that parallel composition of compatible interfaces is monotonic with respect to refinement. Note that Moore and Bidirectional Interfaces force a delay of at least one transition between causally dependent input and output variables, exactly like Moore machines. Reference [60] develops the concept

[31] This formalism is thus not purely synchronous and is mentioned in this section with a slight abuse.
[32] http://www.tinyos.net/

of simulation distances for interfaces, thereby taking robustness issues into account by tolerating errors. Finally, Web services Interfaces were proposed in [41].

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 [142, 12, 49] 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 [142, 137] in order to allow for loose specifications of systems. Since then, they have been considered for automata [140] and formal languages [180, 181] and applied to a wide range of application domains (see [12] 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 5.2 and 5.3. Modal Specifications offer built-in conjunction of specifications [141, 184]. The expressiveness of Modal Specifications has been characterized as a strict fragment of the Hennessy-Milner logic in [49] and also as a strict fragment of the mu-calculus in [104]. The formalism is rich enough to specify safety properties as well as restricted forms of liveness properties. *Modal Interfaces* with a correct notion of compatibility were introduced in [182, 183] and the problem of alphabet equalization with weak and strong alphabet extensions was first correctly addressed in the same references. In [22], 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 [22], i.e., two Modal Interfaces are only compatible if incompatibility between two interfaces cannot occur in any environment. A verification tool called MIO Workbench is available. The quotient of Modal Specifications was studied in [136, 181]. Determinism plays a role in the modal theory. *Non-deterministic Modal Interfaces* have possibly non-deterministic i/o-automata as class of components. Their corresponding computational procedures are of higher complexity than for deterministic ones. A Modal Interface is said to be deterministic if its *may*-transition relation is deterministic. For nondeterministic Modal Interfaces, modal refinement is *incomplete* [140]: there are nondeterministic Modal Interfaces $\mathscr{C}_1$ and $\mathscr{C}_2$ for which the set of implementations of $\mathscr{C}_1$ is included in that of $\mathscr{C}_2$ without $\mathscr{C}_1$ being a modal refinement of $\mathscr{C}_2$. 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 [13, 23] whereas the problem is PTIME-complete if determinism is assumed [184, 24]. The benefits of the determinism assumption in terms of complexity for various decision problems on modal specifications is underlined in [24]. With the aim to preserve deadlock freedom, [52] defines a new refinement relation for modal transition systems (MTS). This refinement "supports itself" e.g. in the sense of thoroughness - in contrast to the standard modal refinement. Finally, the longstanding conflict between unspecified inputs being allowed in Interface Automata but forbidden in MTS is resolved in [50]. The "merge" of non-deterministic Modal Specifications regarded as partial models has been considered in [201]. This operation consists in looking for common refinements of initial specifications and is thus similar to the conjunction operation presented here. In [201, 105], algorithms to compute the maximal

common refinements (which are not unique when non-determinism is allowed) are proposed. They are implemented in the tool MTSA [95]. Assume/guarantee contracts viewed as pairs of Modal Specifications were proposed in [109]. 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 [194], Existential Live Sequence Charts are translated into Modal Specifications, hence providing a mean to specify modal contracts. It was recently shown in [149, 148] that the proposed model of Modal Interface Automata (MIA), a rich subset of Input-Output Modal Transition Systems (IOMTS) [138] featuring explicit output-must-transitions while input-transitions are always allowed implicitly, indeed possesses a conjunction. MIA are not restricted to be deterministic and revisit the model of IOMTS.

Regarding extensions, *Acceptance Interfaces* were proposed by J-B. Raclet [180, 181]. Informally, an Acceptance Interface consists of a set of states, with, for each state, a set of *ready sets*, where a ready set is a set of possible outgoing transitions from that state. Hence, each state of Acceptance Interfaces is labeled with a set of sets of transitions which explicitly specifies its set of possible models. Acceptance Interfaces are more expressive than Modal Interfaces but at the price of a prohibitive complexity for the various relations and operators of the theory. Modal Interfaces have been enhanced with *marked states* by Caillaud and Raclet [25]. 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 [31] to the separate compilation of multiple clocked synchronous programs.

Regarding extensions dealing with time, Timed Automata [7] 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 [8, 9, 38] 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 on top of event-clock automata, seen as corresponding components. A first interface theory able to capture the timing aspects of components is *Timed Interfaces* [86]. 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 [80] fills a gap in the work introduced in [86] by defining a refinement operation. In particular, it is shown that compatibility is preserved by refinement. This theory also proposes a conjunction and a quotient operation and is implemented in the tool Ecdar [81]. Timed Specification Theories are revisited from a linear-time perspective in [69]. The first *timed* extension of modal transition systems was published in [59]. 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 [108]. Another timed extension of Modal Specifications was proposed in [39]. In this formalism, tran-

sitions 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 [36, 37]. [135] addresses the problem of robust implementations in timed specification theories.

Resources other than time were also considered—with energy as the main target. *Resource Interfaces* [64] can be used to enrich a variety of interface formalisms (Interface Automata [84], Assume/Guarantee Interfaces [85], 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. Weighted modal transition systems are proposed in [128, 20], in which each transition is decorated with a weight interval that indicates the range of concrete weight values available to the potential implementations. In this way resource constraints can be modeled using the modal approach. In the same direction, [21] proposes a novel formalism of label-structured modal transition systems that combines the classical may/must modalities on transitions with structured labels that represent quantitative aspects of the model. Last, the issue of contracts for heterogeneous systems is addressed in [144, 145] by building on top of the tag machine component model [27].

Interfaces theories encompassing probability have been more recently proposed. Like the Interval Markov Chain (IMC) formalism [127] they generalize, *Constraint Markov Chains* (CMC) [56] are abstractions of a (possibly infinite) sets of Discrete Time Markov Chains. Instead of assigning a fixed probability to each transition, transition probabilities are kept symbolic and defined as solutions of a set of first order formulas. Variability across implementations is made possible not only with symbolic transition probabilities, but also thanks to the labeling of each state by a set of valuations or sets of atomic propositions. This allows CMCs to be composed thanks to a conjunction and a product operators. While the existence of a residuation operator remains an open problem, CMCs form an interface theory in which satisfaction and refinement are decidable, and compositions can be computed using quantifier elimination algorithms. In particular, CMCs with polynomial constraints form the least class of CMCs closed under all composition operators. In [92], the complexity of several problems for IMCs is studied. The complexity gap for thorough refinement of two IMCs and for deciding the existence of a common implementation for an unbounded number of IMCs is closed by showing that these problems are EXPTIME-complete. *Abstract Probabilistic Automata* (APA) [91] 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.

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 $\mathscr{C}$, of the two sets $\mathcal{E}_{\mathscr{C}}$ and $\mathcal{M}_{\mathscr{C}}$ is new. It clarifies a number of concepts. In particular, the interface theories inherit from the properties of the meta-theory without

the need for specific proofs. The restriction operator for Modal Interfaces is new and so is its use in decomposing a contract into an architecture of sub-contracts. The encoding of Assume/Guarantee reasoning in the framework of Modal Interfaces is also new. Note that this deeply relies on the meta-theory for its justification—inasmuch as the proposed formula provides a valid coding only under specific conditions regarding the tuple $(A_1, \ldots, A_n; G)$. Casting interface theories into the meta-theory was developed for the basic interface theories only. It would be useful to extend this to the different variants and see what the benefit would be. Benoît Caillaud has developed the MICA tool [55], which implements Modal Interfaces with all the operations and services discussed in this section.

6 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.

6.1 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 companion paper [30] demonstrates, using contracts makes semi-formal design safer. Small analysis steps are within the reach of human reasoning. In contrast, lifting a combination of small local reasoning steps to a system-wide analysis—as required when virtually exploring system integration—is difficult and error prone as it involves the risk of wrong circular reasoning. Relying on contracts provides the formal guidance and support for a correct system integration analysis.

Contracts improve requirement engineering: As illustrated in the Parking Garage example of companion paper [30], 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 an "orthogonal" support for all methodologies and can be used in any flow as a supporting technology in composing and refining designs.

6.2 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.

6.3 Status of practice

The use of contract-based techniques in system design is in its infancy in industry. First experiments with this concept showed that following the discipline of making assumptions versus guarantees explicit is by itself already a considerable clarification in requirement engineering, regardless of the formal support provided in addition. Regarding the formal support provided, further maturation is still needed for the formal concepts behind contracts to be well supported by tools and clear enough to be widely accepted by engineers in their day-to-day work. 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.

6.4 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 examplified by our AUTOSAR case study is already a significant help, useful for requirements engineering too. Altogether, a contract engine (such as the MICA tool [55] presented in companion paper [30]) 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. This would provide a smooth transition path to contract based design in practice.

ACKNOWLEDGEMENTS: The authors are indebted to Yishai Feldman, IBM Research, Haifa, for an in-depth reading of a first version of this work. Also, Oguzcan Oguz, Robotics and Mechatronics department, University of Twente, provided helpful remarks and questions regarding A/G-contracts.

References

[1] Martín Abadi and Leslie Lamport. Composing specifications. *ACM Trans. Program. Lang. Syst.*, 15(1):73–132, January 1993.

[2] Martín Abadi and Leslie Lamport. Conjoining specifications. *ACM Trans. Program. Lang. Syst.*, 17(3):507–534, 1995.

[3] Martín Abadi, Leslie Lamport, and Pierre Wolper. Realizable and unrealizable specifications of reactive systems. In Giorgio Ausiello, Mariangiola Dezani-Ciancaglini, and Simona Ronchi Della Rocca, editors, *ICALP*, volume 372 of *Lecture Notes in Computer Science*, pages 1–17. Springer, 1989.

[4] 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.

[5] 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.

[6] 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.

[7] Rajeev Alur and David L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235, 1994.

[8] 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.

[9] 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.

[10] Rajeev Alur, Thomas A. Henzinger, and Orna Kupferman. Alternating-time temporal logic. *J. ACM*, 49(5):672–713, 2002.

[11] 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.

[12] 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.

[13] 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.

[14] Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking*. MIT Press, Cambridge, 2008.

[15] 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.

[16] 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.

[17] 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.

[18] 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.

[19] 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.

[20] Sebastian S. Bauer, Uli Fahrenberg, Line Juhl, Kim G. Larsen, Axel Legay, and Claus R. Thrane. Weighted modal transition systems. *Formal Methods in System Design*, 42(2):193–220, 2013.

[21] Sebastian S. Bauer, Line Juhl, Kim G. Larsen, Axel Legay, and Jirí Srba. Extending modal transition systems with structured labels. *Mathematical Structures in Computer Science*, 22(4):581–617, 2012.

[22] 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.

[23] 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.

[24] Nikola Benes, Jan Kretínský, Kim Guldstrand Larsen, and Jirí Srba. On determinism in modal transition systems. *Theoretical Computer Science*, 410(41):4026–4043, 2009.

[25] Benoît Caillaud and Jean-Baptiste Raclet. Ensuring Reachability by Design. In *Int. Colloquium on Theoretical Aspects of Computing*, sept 2012.

[26] Albert Benveniste and Gérard Berry. The synchronous approach to reactive and real-time systems. *Proceedings of the IEEE*, 79(9):1270–1282, 1991.

[27] Albert Benveniste, Benoît Caillaud, Luca P. Carloni, and Alberto L. Sangiovanni-Vincentelli. Tag machines. In Wayne Wolf, editor, *EMSOFT*, pages 255–263. ACM, 2005.

[28] 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.

[29] Albert Benveniste, Benoit Caillaud, Dejan Nickovic, Roberto Passerone, Jean-Baptiste Raclet, Philipp Reinkemeier, Alberto Sangiovanni-Vincentelli, Werner Damm, Thomas Henzinger, and Kim G. Larsen. Contracts for System Design. Research Report RR-8147, Inria, November 2012. http://hal.inria.fr/hal-00757488.

[30] Albert Benveniste, Benoît Caillaud, Dejan Nickovic, Roberto Passerone, Jean-Baptiste Raclet, Philipp Reinkemeier, Alberto Sangiovanni-Vincentelli, Werner Damm, Tom Henzinger, and Kim Larsen. Contracts for System Design: Methodology and Application cases. Research Report RR-8760, Inria, 2015.

[31] Albert Benveniste, Benoît Caillaud, and Jean-Baptiste Raclet. Application of interface theories to the separate compilation of synchronous programs. In *CDC*, pages 7252–7258. IEEE, 2012.

[32] 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.

[33] Albert Benveniste, Dejan Nickovic, and Thomas Henzinger. Compositional Contract Abstraction for System Design. Research Report RR-8460, Inria, January 2014. http://hal.inria.fr/hal-00938854.

[34] 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.

[35] Gerard Berry. The effectiveness of synchronous languages for the development of safety-critical systems. White paper, Esterel Technologies, 2003.

[36] 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.

[37] 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.

[38] 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.

[39] 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.

[40] Antoine Beugnard, Jean-Marc Jézéquel, and Noël Plouzeau. Making components contract aware. *IEEE Computer*, 32(7):38–45, 1999.

[41] Dirk Beyer, Arindam Chakrabarti, and Thomas A. Henzinger. Web service interfaces. In Allan Ellis and Tatsuya Hagino, editors, *WWW*, pages 148–159. ACM, 2005.

[42] Purandar Bhaduri and S. Ramesh. Interface synthesis and protocol conversion. *Formal Aspects of Computing*, 20(2):205–224, 2008.

[43] 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.

[44] Simon Bliudze and Joseph Sifakis. The Algebra of Connectors - Structuring Interaction in BIP. *IEEE Trans. Computers*, 57(10):1315–1330, 2008.

[45] R. Bloem and B. Jobstmann. Manual for property-based synthesis tool. Technical Report Prosyd D2.2/3, 2006.

[46] 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.

[47] Grady Booch, James Rumbaugh, and Ivar Jacobson. *Unified Modeling Language User Guide, The (2nd Edition) (Addison-Wesley Object Technology Series)*. Addison-Wesley Professional, 2005.

[48] 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.

[49] Gérard Boudol and Kim Guldstrand Larsen. Graphical versus logical specifications. *Theor. Comput. Sci.*, 106(1):3–20, 1992.

[50] Ferenc Bujtor, Sascha Fendrich, Gerald Lüttgen, and Walter Vogler. Nondeterministic modal interfaces. In *41st International Conference on Current Trends in Theory and Practice of Computer Science*, SOFSEM 2015, pages 152–163, Pec pod Sněžkou, Czech Republic, January 24-29, 2015.

[51] Ferenc Bujtor and Walter Vogler. Error-pruning in interface automata. In *40th International Conference on Current Trends in Theory and Practice of Computer Science*, SOFSEM 2014, pages 162–173, Nový Smokovec, Slovakia, January 26-29, 2014.

[52] Ferenc Bujtor and Walter Vogler. Failure semantics for modal transition systems. In *14th International Conference on Application of Concurrency to System Design*, ACSD 2014, pages 42–51, Tunis La Marsa, Tunisia, June 23-27, 2014.

[53] Jerry R. Burch. *Trace Algebra for Automatic Verification of Real-Time Concurrent Systems*. PhD thesis, School of Computer Science, Carnegie Mellon University, August 1992.

[54] Jerry R. Burch, Roberto Passerone, and Alberto L. Sangiovanni-Vincentelli. Overcoming heterophobia: Modeling concurrency in heterogeneous systems. In *Proceedings of the 2nd 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.

[55] Benoît Caillaud. Mica: A Modal Interface Compositional Analysis Library, October 2011. http://www.irisa.fr/s4/tools/mica.

[56] 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.

[57] Daniela Cancila and Roberto Passerone. Functional and structural properties in the Model-Driven Engineering approach. In *Proceedings of the* 13th *IEEE International Conference on Emerging Technologies and Factory Automation*, ETFA08, Hamburg, Germany, September 15–18, 2008.

[58] 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.

[59] 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.

[60] 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.

[61] Arindam Chakrabarti. *A Framework for Compositional Design and Analysis of Systems*. PhD thesis, EECS Department, University of California, Berkeley, Dec 2007.

[62] 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*, pages 428–441, 2002.

[63] 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.

[64] 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.

[65] 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.

[66] 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.

[67] Chris Chilton, Bengt Jonsson, and Marta Kwiatkowska. An algebraic theory of interface automata. *Theoretical Computer Science*, 549:146–174, 2014.

[68] Chris Chilton, Bengt Jonsson, and Marta Z. Kwiatkowska. Compositional assume-guarantee reasoning for input/output component theories. *Sci. Comput. Program.*, 91:115–137, 2014.

[69] 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.

[70] E. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 1999.

[71] Edmund M. Clarke, David E. Long, and Kenneth L. McMillan. Compositional model checking. In *LICS*, pages 353–362, 1989.

[72] Patrick Cousot and Radhia Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In Robert M. Graham, Michael A. Harrison, and Ravi Sethi, editors, *POPL*, pages 238–252. ACM, 1977.

[73] Patrick Cousot and Radhia Cousot. Abstract interpretation and application to logic programs. *J. Log. Program.*, 13(2&3):103–179, 1992.

[74] Patrick Cousot and Radhia Cousot. Abstract Interpretation Frameworks. *J. Log. Comput.*, 2(4):511–547, 1992.

[75] Przemyslaw Daca, Thomas Henzinger, Willibald Krenn, and Dejan Nickovic. Compositional specifications for ioco testing. In *ICST*, 2014.

[76] 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.

[77] Werner Damm and David Harel. LSCs: Breathing life into message sequence charts. *Formal Methods in System Design*, 19(1):45–80, 2001.

[78] Werner Damm, Angelika Votintseva, Alexander Metzner, Bernhard Josko, Thomas Peikenkamp, and Eckard Bde. Boosting reuse of embedded automotive applications through rich components. In *Proceedings of FIT 2005 - Foundations of Interface Technologies*, 2005.

[79] Abhijit Davare, Douglas Densmore, Liangpeng Guo, Roberto Passerone, Alberto L. Sangiovanni-Vincentelli, Alena Simalatsar, and Qi Zhu. METROII: A design environment for cyber-physical systems. *ACM Transactions on Embedded Computing Systems*, 12(1s):49:1–49:31, March 2013.

[80] 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.

[81] 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.

[82] 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.

[83] 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.

[84] 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.

[85] 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.

[86] 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.

[87] Benoît Delahaye. *Modular Specification and Compositional Analysis of Stochastic Systems*. PhD thesis, Université de Rennes 1, 2010.

[88] 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.

[89] 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.

[90] 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.

[91] 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.

[92] Benoît Delahaye, Kim G. Larsen, Axel Legay, Mikkel L. Pedersen, and Andrzej Wasowski. Consistency and refinement for interval markov chains. *J. Log. Algebr. Program.*, 81(3):209–226, 2012.

[93] Douglas Densmore, Roberto Passerone, and Alberto L. Sangiovanni-Vincentelli. A platform-based taxonomy for ESL design. *IEEE Design and Test of Computers*, 23(5):359–374, May 2006.

[94] David L. Dill. *Trace Theory for Automatic Hierarchical Verification of Speed-Independent Circuits*. ACM Distinguished Dissertations. MIT Press, 1989.

[95] 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.

[96] 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.

[97] Dumitru Potop-Butucaru and Stephen Edwards and Gérard Berry. *Compiling Esterel*. Springer V., 2007. ISBN: 0387706267.

[98] 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.

[99] 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.

[100] 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.

[101] 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.

[102] Orlando Ferrante, Roberto Passerone, Alberto Ferrari, Leonardo Mangeruca, and Christos Sofronis. BCL: a compositional contract language for embedded systems. In *Proceedings of the* 19^{th} *IEEE International Conference on Emerging Technologies and Factory Automation*, ETFA14, Barcelona, Spain, September 16–19, 2014.

[103] Orlando Ferrante, Roberto Passerone, Alberto Ferrari, Leonardo Mangeruca, Christos Sofronis, and Massimiliano D'Angelo. Monitor-based run-time contract verification of distributed systems. In *Proceedings of the* 9^{th} *IEEE International Symposium on Industrial Embedded Systems*, SIES14, Pisa, Italy, June 18–20, 2014.

[104] G. Feuillade. Modal specifications are a syntactic fragment of the Mu-calculus. Research Report RR-5612, INRIA, June 2005.

[105] 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.

[106] Frédéric Boussinot and Robert de Simone. The Esterel language. *Proceedings of the IEEE*, 79(9):1293–1304, 1991.

[107] Peter Fritzson. *Principles of Object-Oriented Modeling and Simulation with Modelica 2.1*. Wiley, 2003.

[108] 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.

[109] 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.

[110] Susanne Graf, Roberto Passerone, and Sophie Quinton. Contract-based reasoning for component systems with rich interactions. In Alberto L. Sangiovanni-Vincentelli, Haibo Zeng, Marco Di Natale, and Peter Marwedel, editors, *Embedded Systems Development: From Functional Models to Implementations*, volume 20 of *Embedded Systems*, chapter 8, pages 139–154. Springer New York, 2014.

[111] 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.

[112] Orna Grumberg and David E. Long. Model checking and modular verification. *ACM Trans. Program. Lang. Syst.*, 16(3):843–871, 1994.

[113] 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.

[114] Nicolas Halbwachs. *Synchronous programming of reactive systems*. Kluwer Academic Pub., 1993.

[115] 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.

[116] 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.

[117] 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.

[118] David Harel. Statecharts: A visual formalism for complex systems. *Sci. Comput. Program.*, 8(3):231–274, 1987.

[119] 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.

[120] 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.

[121] David Harel, Assaf Marron, and Gera Weiss. Behavioral programming. *Commun. ACM*, 55(7):90–100, 2012.

[122] 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.

[123] 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.

[124] Thomas A. Henzinger and Dejan Nickovic. Independent implementability of viewpoints. In Radu Calinescu and David Garlan, editors, *Monterey Workshop*, volume 7539 of *Lecture Notes in Computer Science*, pages 380–395. Springer, 2012.

[125] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, 1969.

[126] Cliff B. Jones. Specification and design of (parallel) programs. In *IFIP Congress*, pages 321–332, 1983.

[127] B. Jonsson and K. G. Larsen. Specification and refinement of probabilistic processes. In *Logic in Computer Science (LICS)*, pages 266–277. IEEE Computer, 1991.

[128] Line Juhl, Kim G. Larsen, and Jirí Srba. Modal transition systems with weight intervals. *J. Log. Algebr. Program.*, 81(4):408–421, 2012.

[129] Gilles Kahn. The Semantics of Simple Language for Parallel Programming. In *IFIP Congress*, pages 471–475, 1974.

[130] S. Karris. *Introduction to Simulink with Engineering Applications*. Orchard Publications, 2006.

[131] Gabor Karsai, Janos Sztipanovitz, Akos Ledczi, and Ted Bapty. Model-integrated development of embedded software. *Proceedings of the IEEE*, 91(1), January 2003.

[132] 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.

[133] Leslie Lamport. Proving the correctness of multiprocess programs. *IEEE Trans. Software Eng.*, 3(2):125–143, 1977.

[134] C. Larman and V.R. Basili. Iterative and incremental developments: a brief history. *Computer*, 36(6):47–56, June 2003.

[135] Kim G. Larsen, Axel Legay, Louis-Marie Traonouez, and Andrzej Wasowski. Robust synthesis for real-time systems. *Theor. Comput. Sci.*, 515:96–122, 2014.

[136] 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.

[137] 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.

[138] 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.

[139] 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.

[140] 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.

[141] 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.

[142] 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.

[143] H. T. T. Le and R. Passerone. Refinement-based synthesis of correct contract model decompositions. In *Proceedings of the* 12^{th} *ACM-IEEE International Conference on Formal Methods and Models for System Design*, MEMOCODE14, Lausanne, Switzerland, October 19–21, 2014.

[144] Hoa Thi Thieu Le, Roberto Passerone, Uli Fahrenberg, and Axel Legay. A tag contract framework for heterogeneous systems. In *Proceedings of the* 12^{th} *International Workshop on Foundations of Coordination Languages and Self Adaptive Systems*, FOCLASA13, Malaga, Spain, September 11, 2013.

[145] Hoa Thi Thieu Le, Roberto Passerone, Uli Fahrenberg, and Axel Legay. Tag machines for modeling heterogeneous systems. In *Proceedings of the* 13^{th} *International Conference on Application of Concurrency to System Design*, ACSD13, pages 186–195, Barcelona, Spain, July 8–10, 2013.

[146] 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.

[147] 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.

[148] Gerald Lüttgen and Walter Vogler. Modal interface automata. *Logical Methods in Computer Science*, 9(3), 2013.

[149] Gerald Lüttgen and Walter Vogler. Richer interface automata with optimistic and pessimistic compatibility. *ECEASST*, 66, 2013.

[150] 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.

[151] Nancy A. Lynch and Eugene W. Stark. A proof of the kahn principle for input/output automata. *Inf. Comput.*, 82(1):81–92, 1989.

[152] Zohar Manna and Amir Pnueli. *Temporal verification of reactive systems: Safety*. Springer, 1995.

[153] 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.

[154] 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.

[155] B. Meyer. Applying "design by contract". *IEEE Computer*, 25(10):40–51, October 1992.

[156] Bertrand Meyer. *Touch of Class: Learning to Program Well Using Object Technology and Design by Contract*. Springer, Software Engineering, 2009.

[157] Antoine Miné. Weakly Relational Numerical Abstract Domains. Phd, Ecole Normale Supérieure, département d'informatique, Dec 2004. http://www.di.ens.fr/~mine/these/these-color.pdf.

[158] M.W. Maier. Architecting Principles for Systems of Systems. *Systems Engineering*, 1(4):267–284, 1998.

[159] Walid A. Najjar, Edward A. Lee, and Guang R. Gao. Advances in the dataflow computational model. *Parallel Computing*, 25(13-14):1907–1929, 1999.

[160] Radu Negulescu. *Process Spaces and the Formal Verification of Asynchronous Circuits*. PhD thesis, University of Waterloo, Canada, 1998.

[161] 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.

[162] 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.

[163] Object Management Group (OMG). Unified Modeling Language (UML) specification. [online], http://www.omg.org/spec/UML/.

[164] Object Management Group (OMG). A UML profile for MARTE, beta 1. OMG Adopted Specification ptc/07-08-04, OMG, August 2007.

[165] Object Management Group (OMG). System modeling language specification v1.1. Technical report, OMG, 2008.

[166] Object constraint language, version 2.0. OMG Available Specification formal/06-05-01, Object Management Group, May 2006.

[167] The Design Automation Standards Committee of the IEEE Computer Society, editor. *1850-2010 - IEEE Standard for Property Specification Language (PSL)*. IEEE Computer Society, 2010.

[168] 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.

[169] 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.

[170] Roberto Passerone, Luca de Alfaro, Thomas A. Henzinger, and Alberto L. Sangiovanni-Vincentelli. Convertibility verification and converter synthesis: Two faces of the same coin. In *Proceedings of the* 20^{th} *IEEE/ACM International Conference on Computer-Aided Design (ICCAD02)*, pages 132–139, San Jose, California, November 10–14, 2002. IEEE Computer Society, Los Alamitos, CA, USA.

[171] 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.

[172] 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.

[173] Paul Le Guernic and Thierry Gautier and Michel Le Borgne and Claude Le Maire. Programming real-time applications with Signal. *Proceedings of the IEEE*, 79(9):1321–1336, 1991.

[174] 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.

[175] Alessandro Pinto, Alvise Bonivento, Alberto L. Sangiovanni-Vincentelli, Roberto Passerone, and Marco Sgroi. System level design paradigms: Platform-based design and communication synthesis. *ACM Transactions on Design Automation of Electronic Systems*, 11(3):537–563, July 2006.

[176] Dumitru Potop-Butucaru, Benoît Caillaud, and Albert Benveniste. Concurrency in synchronous systems. *Formal Methods in System Design*, 28(2):111–130, 2006.

[177] 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.

[178] Terry Quatrani. *Visual modeling with Rational Rose 2000 and UML (2nd ed.)*. Addison-Wesley Longman Ltd., Essex, UK, UK, 2000.

[179] 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.

[180] Jean-Baptiste Raclet. *Quotient de spécifications pour la réutilisation de composants*. PhD thesis, Ecole doctorale Matisse, université de Rennes 1, November 2007.

[181] Jean-Baptiste Raclet. Residual for Component Specifications. In *Proc. of the 4th International Workshop on Formal Aspects of Component Software (FACS'07)*, 2007.

[182] 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.

[183] 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.

[184] 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.

[185] Jan Reineke and Stavros Tripakis. Basic problems in multi-view modeling. In *TACAS*, 2014.

[186] Philipp Reinkemeier and Ingo Stierand. Compositional timing analysis of real-time systems based on resource segregation abstraction. In Gunar Schirner, Marcelo Götz, Achim Rettberg, Mauro Cesar Zanella, and Franz J. Rammig, editors, *Embedded Systems: Design, Analysis and Verification - 4th IFIP TC 10 International Embedded Systems Symposium, IESS 2013, Paderborn, Germany, June 17-19, 2013. Proceedings*, volume 403 of *IFIP Advances in Information and Communication Technology*, pages 181–192. Springer, 2013.

[187] Philipp Reinkemeier and Ingo Stierand. Real-Time Contracts - A Contract Theory Considering Resource Supplies and Demands. Reports of SFB/TR 14 AVACS 100, SFB/TR 14 AVACS, July 2014. http://www.avacs.org.

[188] 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.

[189] Robert W. Floyd. Assigning meaning to programs. In J.T. Schwartz, editor, *Proceedings of Symposium on Applied Mathematics*, volume 19, pages 19–32, 1967.

[190] A. Sangiovanni-Vincentelli, S. Shukla, J. Sztipanovits, G. Yang, and D. Mathaikutty. Metamodeling: An emerging representation paradigm for system-level design. *IEEE Design and Test of Computers*, 26(3):54–69, May/June 2009.

[191] 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.

[192] Alberto Sangiovanni-Vincentelli, Werner Damm, and Roberto Passerone. Taming Dr. Frankenstein: Contract-based design for cyber-physical systems. *European Journal of Control*, 18(3):217–238, 2012.

[193] D. Schmidt. Model-driven engineering. *IEEE Computer*, pages 25–31, February 2006.

[194] German Sibay, Sebastian Uchitel, and Víctor Braberman. Existential Live Sequence Charts Revisited,. In *ICSE 2008: 30th International Conference on Software Engineering*. ACM, May 2008.

[195] 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.

[196] 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.

[197] Ingo Stierand, Philipp Reinkemeier, and Purandar Bhaduri. Virtual integration of real-time systems based on resource segregation abstraction. In *Formal Modeling and Analysis of Timed Systems - 12th International Conference*, FORMATS 2014, pages 206–221, Florence, Italy, September 8-10, 2014.

[198] Ingo Stierand, Philipp Reinkemeier, Tayfun Gezgin, and Purandar Bhaduri. Real-time scheduling interfaces and contracts for the design of distributed embedded systems. In *8th IEEE International Symposium on Industrial Embedded Systems*, SIES 2013, pages 130–139, Porto, Portugal, June 19-21, 2013.

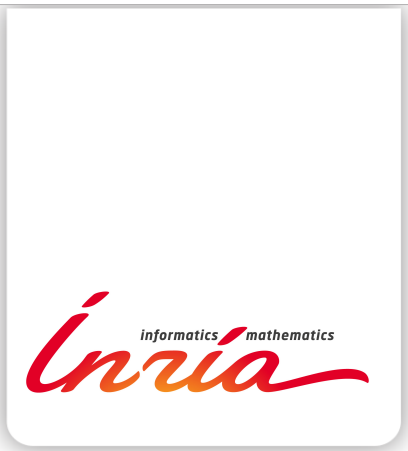
[199] Stavros Tripakis, Ben Lickly, Thomas A. Henzinger, and Edward A. Lee. On relational interfaces. In *Proc. of the 9th ACM & IEEE International conference on Embedded software (EMSOFT'09)*, pages 67–76. ACM, 2009.

[200] 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.

[201] 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.

[202] 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.

[203] Elizabeth S. Wolf. *Hierarchical Models of Synchronous Circuits for Formal Verification and Substitution*. PhD thesis, Department of Computer Science, Stanford University, October 1995.



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