

Contracts for the Design of Embedded Systems

Part II: Theory

Albert Benveniste[†], Jean-Baptiste Raclet*,
 Benoît Caillaud[†], Dejan Nickovic[§], Roberto Passerone[‡], Alberto Sangiovanni-Vincentelli^{||}
 Tom Henzinger[§], and Kim Larsen[¶]

[†]INRIA/IRISA, Rennes, France. corresp. author: Albert.Benveniste@inria.fr

[‡]University of Trento, Trento, Italy [§]IST Austria, Klosterneuburg [¶]Aalborg University

*IRIT, Toulouse, France ||University of California at Berkeley

Abstract—This is Part II of a sequence of two papers on *Contract-based Design (CBD)*.

Part I introduced concepts, presented how CBD addresses the challenges facing designers of large-scale complex systems, placed it in the context of existing design methodologies and showed how it can be used effectively together with any other methodology so far proposed.

Part II covers the theoretical foundations that are essential to make CBD robust and well supported by tools and software frameworks. We believe that contract-based design is going to be a key part of the future of system design and for this reason, this contribution is important for the design community.

CONTENTS

I	Introduction	3			
II	Recalling the Parking Garage Example	3			
	II-A System-level Specifications	3			
	II-B Defining, firming up, and executing requirements	3			
	II-C Identifying responsibilities, partitioning and allocating requirements	4			
	II-D Further Refining Requirements	5			
	II-E Fundamental Properties	5			
	II-F Requirements and architecture	5			
III	A Mathematical Meta-theory of Contracts	6			
	III-A Independent implementability	6			
	III-A1 Abstraction	7			
	III-A2 Compatibility	7			
	III-A3 The notion of refinement	8			
	III-B The Meta-Theory of Contracts	8			
	III-B1 Implementation	8			
	III-B2 Contracts	8			
	III-B3 Refinement	8			
	III-B4 Conjunction	8			
	III-B5 Product	9			
	III-B6 Consistency	9			
	III-B7 Compatibility	10			
	III-B8 Quotient	10			
	III-C Discussion	10			
	III-D Casting vertical contracts in the meta-theory	10			
	III-E Conclusions	11			
IV	Observers as Contracts	12			
	IV-A Tests	12			
	IV-B Observers	12			
	IV-C Observers as a Theory of Contracts	13			
	IV-D Implementing observers	13			
	IV-D1 Using Simulink/Scade/Signal	13			
	IV-D2 Using PSL	14			
	IV-D3 Using LSCs and Play Engines	15			
	IV-E Discussion	16			
V	Assume/Guarantee contracts	16			
	V-A Assume/Guarantee contracts with strong/weak assumptions	17			
	V-B Handling dissimilar alphabets	18			
	V-C The framework and how it instantiates the meta-theory	18			
	V-D Answering the questions raised in Section II on the parking garage example	19			
	V-E Practical implementation	19			
	V-F The problem with handling dissimilar alphabets	20			
	V-G Extensions	20			
	V-G1 Time	20			
	V-G2 Stochastic	20			
	V-H Discussion	20			
VI	Interfaces as Contracts	21			
	VI-A Interfaces and Modalities	21			
	VI-B The framework and how it instantiates the meta-theory	22			
	VI-C Handling dissimilar alphabets	24			
	VI-D Practical implementation	25			
	VI-E Expressing contracts with assumptions and guarantees as Modal Interfaces	25			

This work was funded in part by the European STREP-COMBEST project number 215543, the European project CESAR of the ARTEMIS Joint Undertaking, the Artist Design Network of Excellence number 214373, the MARCO FCRP MuSyC and GSRC grants, the DARPA METAII Project, the VKR Center of Excellence MT-LAB, and the German Innovation Alliance on Embedded Systems SPES2020.

VI-F	Refinements and Extensions	26
VI-F1	A refinement of Modal Inter- faces: Acceptance Interfaces	26
VI-F2	Time	26
VI-F3	Synchronicity	26
VI-F4	Resources	27
VI-F5	Stochastic	27
VI-G	Discussion	27
VII	Concluding Remarks	27
References		29

*

I. INTRODUCTION

In a companion paper [16], we introduced the concepts underlying Contract-Based Design and its use in a variety of scenarios. We presented the challenges facing designers of large-scale complex systems today including management of complexity, management of multi-tier supply and design chains and requirements capture and engineering. Then we reviewed the design methodologies and design concepts so far used to address these challenges (viewpoints, layered design, component-based design, the V-model of design, model-based development, virtual integration, platform-based design) and showed how contracts can be used in all these approaches to enhance their applicability and the quality of the final result. Finally, we examined three examples of the use of contracts: one simple example related to requirement engineering to underline the key steps of contract-based design, one about the use of contracts in enhancing and extending the AUTOSAR automotive standard and one related to analog-mixed signal integrated circuit design. Central to contracts was the distinction between assumptions and guarantees, mirroring the different roles and responsibilities in the design of systems.

In this paper, we develop and review the needed mathematically sound theory underpinning the use of contracts in general terms that subsumes the assume-guarantee paradigm presented in [16]. We first briefly recall in Section II the Parking Garage example of [16]; this example is used to illustrate several technical difficulties that contract-based design has to face. Section III contains the main contribution of this paper, a mathematical “meta-theory” of contracts. The central issue of independent implementability is pinpointed in Section III-A. The meta-theory is developed in Section III-B, thus collecting requirements for any valid theory of contracts or interfaces. Section III-D explains how the so-called “vertical contracts” extensively discussed in [16] are captured. We then review and contrast a number of candidate instantiations of the meta-theory. We discuss observers as contracts in Section IV. Then we present the approach based on assume/guarantee contracts proposed by the SPEEDS project in Section V, and Interface contract model in Section VI. Finally, an assessment of contract-based design and its future directions are presented in Section VII.

II. RECALLING THE PARKING GARAGE EXAMPLE

In this section we present the essentials of the Parking Garage example, which was fully developed in companion paper [16]. We begin directly with the system-level specification corresponding to the system architecture shown on Figure 1.

A. System-level Specifications

The system-level specification includes requirements concerning the system and its environment (the “user”).

Specification 1 (System level requirements):

- 1) A user may enter only if the entry gate is open;
- 2) The entry gate may open only if the parking is not full;

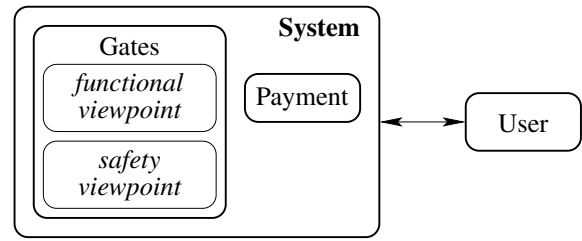


Figure 1. Parking system top-level architecture. See Section II-F regarding functional and safety viewpoints for the Parking.

- 3) If the user presses the button, then the entry gate may open and must return a ticket marked with the time of entrance;
- 4) The cost of parking is a monotonic function of the time spent in the garage;
- 5) Parking is not allowed for more than 3 days;
- 6) The user may pay only after having introduced her ticket;
- 7) Only tickets issued by the gates can be inserted in the payment machine and tickets can be inserted at most once;
- 8) The user can pay only with coins;
- 9) The payment machine only accepts coins or credit cards;
- 10) If the payment is correct, a clearance ticket is returned;
- 11) Only clearance tickets issued by the payment machine can be inserted in the exit gate and clearance tickets can be inserted at most once;
- 12) The clearance ticket is valid for 5 minutes;
- 13) Upon entering a valid clearance ticket, the exit gate opens;
- 14) A user may exit only if the exit gate is open.

Specification 1 can be allocated to two suppliers — one in charge of the gates and one the payment machines, respectively stipulated as usage rules for users. The consideration of requirements 8 and 9 may seem strange; it is, however, motivated by the discussion in Section II-E.

B. Defining, firming up, and executing requirements

Since the top-level specification is often informal, gathering system level requirements typically represent the first step of the design flow. Requirements may not faithfully represent the designer’s intent and thus cannot be taken as “the right definition of the system”, they must be validated with the objective of stating *how good the stated requirements are in reflecting the designer’s (and users’) intent*.

In particular, the process of requirement capture and of writing requirement documents are important matters for discussions with certification bodies. These bodies would typically assess a number of quality criteria for example, selecting them from the following list elaborated by INCOSE, see [82]: Accuracy, Affordability, Boundedness, Class, Complexity, Completeness, Conciseness, Conformance, Consistency, Correctness, Criticality, Level, Orthogonality, Priority,

Risk, Unambiguousness, and Verifiability. There are several specific means that can be used to evaluate these criteria.

To analyze the requirements and verify their consistency and coverage, the informal textual description should be converted into a mathematical representation. A mathematical representation allows *executing* the requirements. Executing requirements is necessary for exploring instances and behaviors of these events, actions, and parameters that meet the constraints imposed by the requirements.

C. Identifying responsibilities, partitioning and allocating requirements

Identifying *responsibilities* in the requirements, i.e., which actor is supposed to guarantee that the requirements are met and which actor impacts and is impacted by the requirement, allows identifying some implicit assumptions that may become a source of faulty design. Table I identifies, for each requirement, which subsystem is responsible for satisfying it and which are affected and affect the requirement. The subsystems involved in the table are “user”, “payment”, and “gates”.

Requirement	Responsible	Involved
R _{1.1}	user	gates
R _{1.2}	gates	
R _{1.3}	gates	user
R _{1.4}	payment	user
R _{1.5}	user	
R _{1.6}	payment	user
R _{1.7}	user	payment
R _{1.8}	user	payment
R _{1.9}	payment	user
R _{1.10}	payment	user
R _{1.11}	user	gates
R _{1.12}	gates	user
R _{1.13}	gates	user
R _{1.14}	user	gates

Table I

Identifying subsystem (including the “user”) responsible for and the ones involved with the requirements. Notation R_{1,i} refers to the *i*th requirement among Specification 1.

To allocate requirements to a subsystem, e.g., “payment”, we proceed as follows, using Table I:

- Collect from Table I the requirements that are involved with “payment” but are not under its responsibility — these are R_{1.7} and R_{1.8}. These are not requirements for the “payment” subsystem. Instead, they constitute the *assumptions* under which the “payment” subsystem must operate.
- Collect the requirements that are under the responsibility of “payment” — these are R_{1.4}, R_{1.6}, R_{1.9}, and R_{1.10}. They constitute the obligations for the “payment” subsystem and are taken as subsystem requirements. To highlight the symmetry with respect to the assumptions when dealing with contracts, the obligations above are often called *guarantees*. We will stick to this wording in the the rest of the paper.

Performing this operation yields the following set of assumptions and guarantees, thus constituting the *contract* for the “payment” subsystem. In this and the following contracts, “*guarantees*” are in italics, whereas “*assumptions*” are in roman:

Contract 1 (Subsystem “Payment”):

- 4) *The cost of staying-in-parking is proportional to the time spent;*
- 6) *The user may pay only after having introduced her ticket;*
- 7) Only tickets issued by the gates can be inserted in the payment machine and tickets can be inserted at most once;
- 8) The user pays with coins;
- 9) *The payment machine only accepts coins or credit cards;*
- 10) *If payment is correct, a clearance ticket is returned.*

Contract 2 (subsystem “Gates”):

- 1) A user may enter only if the entry gate is open;
- 2) *The entry gate may open only if the parking is not full;*
- 3) *If the user presses the button, then the entry gate may open and then must return a ticket marked with the time of entrance;*
- 11) Only clearance tickets issued by the payment machine can be inserted in the exit gate and clearance tickets can be inserted at most once;
- 12) *The clearance ticket is valid for 5 minutes;*
- 13) *Upon entering the clearance ticket, the exit gate opens;*
- 14) A user may exit only if the exit gate is open.

At this point 2 completes the subsystems’ requirements. The last step consists of deriving the user’s requirements:

Contract 3 (“User”):

- 1) *A user may enter only if the entry gate is open;*
- 3) If the user presses the button, then the entry gate may open and then must return a ticket marked with the time of entrance;
- 4) The cost of staying-in-parking is proportional to the time spent;
- 5) *Staying-in-parking is not allowed for more than 3 days;*
- 6) The user may pay only after having introduced her ticket;
- 7) *Only tickets issued by the gates can be inserted in the payment machine and tickets can be inserted at most once;*
- 8) *The user pays with coins;*
- 9) The payment machine only accepts coins or credit cards;
- 10) If payment is correct, a clearance ticket is returned;
- 11) *Only clearance tickets issued by the payment machine can be inserted in the exit gate and clearance tickets can be inserted at most once;*
- 12) The clearance ticket is valid for 5 minutes;
- 13) Upon entering the clearance ticket, the exit gate opens;
- 14) *A user may exit only if the exit gate is open.*

Assumption 1.7 expresses that the user cannot cheat by falsifying a ticket in some way. It occurs as a guarantee in Contract 3 for the “user”. It is tempting to consider that starting with no such assumption as a first step, and then

adding it in a second step after further thinking, is a correct process. It is, however, not! Indeed, as we shall later see, it is problematic to strengthen the assumptions made on the environment, when updating the requirements for a subsystem (here: the “payment”). Problems may arise at integration level if this strengthening of assumptions is not forwarded to the other subsystems with which the considered subsystem is interacting (here: the “user”).

D. Further Refining Requirements

Suppose that the system designer (an OEM) has subcontracted the subsystem “gates” to some supplier for its design. Based on her experience in designing parking management systems, this supplier has an off-the-shelf design implementing a set of assumptions and guarantees consisting of Contract 2 augmented with the following additional assumptions and guarantees (we use new labels for these, while keeping the same convention as before regarding assumptions versus guarantees):

Contract 4 (Subsystem “Off-the-shelf gates”): This contract consists of Contract 2 augmented with the following assumptions and guarantees:

- 1) *If the user presses the button, the entry gate must open within 10 sec;*
- 2) *Upon entering the clearance ticket, the exit gate must open within 10 sec;*
- 3) *When the exit gate opens, the user must exit within 10 sec;*
- 4) *Entry and exit gates are soft enough not to hurt pedestrians.*

The first two additional guarantees are related to the timing behavior of the gates. The new assumption specifies the timing behavior of the user. Finally, the last guarantee is related to the parking hardware. Since the supplier has achieved more with this off-the-shelf parking management system, the corresponding design can be delivered to the OEM, or *at least as it seems to be the case...*

E. Fundamental Properties

At the beginning of this section we mentioned a number of properties that qualify the goodness of the requirements document, these were referred to as INCOSE criteria. A number of these criteria are clearly subject to human judgement. Examples of such properties include: Accuracy, Affordability, Level, Priority. Other INCOSE properties are more essential in that they qualify the soundness of both the requirements document and system design flow. We discuss these properties next by posing them as answering a specific set of questions.

Property 1 (Completeness): *Are the requirements listed in Specification 1 complete?*

Completeness should be defined according to some reference. However, by definition, such a reference does not exist since Specification 1 is the entry point of the design process. Consequently, this question cannot be really formalized in our

context. Still, completeness can be explored in practice by executing and analyzing requirements.

The second question concerns what it means for a design to satisfy a contract.

Property 2 (Correctness): *What does it mean for an actual design to satisfy a given contract?*

The basic idea is that the design should “satisfy all requirements”. While this idea conforms to our proposal that requirements belonging to the same document are composed in a conjunctive way, this is not sufficient. We must in particular handle *assumptions* and *guarantees* differently. A valid understanding of correctness is thus the following: When a design is inserted in an environment that includes other components of the design as well as external agents, if its interactions with the environment satisfy the assumptions, then the component is such that its guarantees to the environment are met.

The next property to be checked concerns the assignment of requirements to subsystems: In which sense can we ensure that replacing system wide Specification 1 by the composition of Contracts 1 to 3 is a safe step? Also, was the supplier for the gates right in proposing her off-the-shelf design? These are important questions that are typically formulated as follows:

Property 3 (Refinement):

- 1) *Does the composition of Contracts 1 to 3 refine Specification 1?*
- 2) *Does Contract 4 for the off-the-shelf gates refine the original Contract 2 for the gates?*

Observe that this refinement property resembles the informal request of “completeness” that was stated as Property 1. The difference here is that system-wide requirements provide the missing reference against which refinement can be assessed.

Each contract sets obligations on the considered subsystem (the requirements in italics) and implicit obligations on the environment (the assumptions made by the subsystem, written in roman). Are these feasible obligations? Note that the same issue arises when we consider the composition of contracts 1 to 3. This property is generally referred to as “consistency”. We also consider its symmetric property of “compatibility” since we distinguished guarantees (to which the subsystem is committed) from assumptions (which are the duty of the environment).

Property 4 (Consistency and Compatibility): *Do designs exist that meet Contract 1-4 consistency? Do environments exist that meet Contract 1-4 compatibility?*

For example, requirements $R_{1.8}$ and $R_{1.9}$ might have been a source of non compatibility between “payment” and “user” — they are, however, compatible as they stand.

F. Requirements and architecture

While refining requirements during the design flow, we may encounter contracts that involve multiple viewpoints, for example reliability of the design. As an example, consider the following *safety* (or *reliability*) viewpoint for the gates:

Contract 5 (subsystem “Gates”, safety viewpoint):

- 1) The probability that a user will cause damage to the gates is less than 10^{-10} ;
- 2) *If the parking is not full, the probability that the entry gate remains closed is less than 10^{-9} ;*
- 3) *If a clearance ticket is entered, the probability that the exit gate remains closed is less than 10^{-9} .*

The question arises of how to deal with requirements that involve assumptions and guarantees that relate to different aspects of the design and yet they are to be considered somehow together.

Question 1 (Conjunction): What does it mean to structure a contract into a set of viewpoints?

An important issue related to viewpoints is to understand what should be the meaning of their conjunction. Referring to our running example, we need to deal with the function realized by the gates (Contract 2, denoted by C_{park}) and its safety requirements (Contract 5, denoted by $C_{\text{park_safety}}$). What should their conjunction $C_{\text{park_all_viewpoints}} = C_{\text{park}} \wedge C_{\text{park_safety}}$ guarantee? If users respect the rules of usage (no cheating with tickets, no damaging), then all guarantees must be met. If, however, users do not cause damage but may cheat with tickets, then only the safety guarantees will be met. Vice-versa, if users do not cheat with tickets but may cause damage, then only the functional guarantees will be guaranteed. The important point here is that functional and safety viewpoints must be kept autonomously, considering them jointly should not cause spurious interaction between them. Only necessary interactions should remain, corresponding to shared states, events or messages, between their respective contracts.

The next issue is of paramount importance, since it relates the subsystems requirements to the systems requirements:

Question 2 (Composition): What is the meaning of assigning contracts to different subsystems?

Contracts 1 and 2 are attached to the different subsystems (not including the “user”). The intent is that these subsystems will be composed to form the whole system. How to mirror the composition of subsystems into a corresponding *composition of contracts*? It should be clear that this new type of composition differs from the conjunction of contracts, for a same subsystem or component.

The various questions raised above will be discussed in Section V-D.

III. A MATHEMATICAL META-THEORY OF CONTRACTS

We argued in the companion paper the relevance of contracts in the design of complex systems and we summarized in the previous section the properties and the open questions related to the use and management of contracts. In this section, we present a novel approach to introducing the mathematical underpinning of contract-based design. In particular, we believe that an abstract approach exposes best the fundamentals and provides guidelines to derive a family of contract-based specific methodologies. Surprisingly, we found that some

existing approaches to verification and design fall into the category of contract-based design if the abstract definition is used.

We begin this section by introducing the notion of independent implementability, an essential concern in compositional methods for system level design, and how it relates to abstraction and refinement. Independent implementability provides an ideal introduction to the meta-theory of contracts, the main contribution of this paper.

A. Independent implementability

In design methodologies such as Platform-Based Design [117], complex systems are developed from components, using a combination of bottom-up and top-down design techniques. For example, bottom-up design is used when building a system by assembling (possibly off-the-shelf) components from a library. Top-down design is used when starting with top-level system-wide requirements and refining them into requirements for subsequent development stages. Entities for reuse are generically referred to as “components” and are exposed to the user through an “interface”. Compositional property checking is useful for both bottom-up and top-down compositional design. In top-down compositional design, the designer decomposes a system-level specification into several sub-system specifications and then submits each sub-system specification for independent development. In bottom-up compositional design, ideally verifying that the assembly of pre-defined components will work as specified should be based on the only basis of analyzing the composition of (abstract) interfaces for each component. Both cases call for *independent implementability*, which is the capability of developing specifications for different sub-systems, independently. In this section we analyze this central concept.

The notions of component, interface, and independent implementability were discussed by de Alfaro and Henzinger [53]. where components are distinguished from interfaces: (quoted from [53]) “A component description answers the following question about a system or software module: *What does it do?* An interface description answers the question *How can it be used?*”. Interfaces differ from components in that (again from [53]) “components do not constrain the environment, whereas interfaces do”. The above distinction may seem somehow casuistic. Still, it brings the authors of [53] to an in-depth development regarding the two instrumental notions of abstraction and compatibility:

- To avoid dealing with excessive details, abstractions must be used;
- Compatibility of a set of modules must be ensured prior to composing them;
- Handling both concepts jointly raises a number of non-trivial issues related to independent implementability.

To discuss this, we take the point of view that system or software modules define constraints on the entities they own: they define a set of ports, actions, and variables; they define types; they constrain behaviors; they may constrain their

environment. The following is an example:

$$S_1 : \left\{ \begin{array}{l} \text{ports: } \left\{ \begin{array}{l} \text{inputs: } x, y \\ \text{outputs: } z \end{array} \right. \\ \text{types: } \left\{ \begin{array}{l} x, y, z \in \mathbb{N} \\ \wedge x \in \text{Distance} \\ \wedge y \in \text{Time} \\ \wedge z \in \text{Speed} \end{array} \right. \\ \text{behaviors: } y \neq 0 \wedge z = x/y \end{array} \right.$$

The partition of ports into inputs and outputs has a topological effect, since it comes with the usual connection rule for blocks, namely: outputs of a block can only be connected to inputs of other blocks. It also induces, by convention, the following additional constraint:

addendum to S_1 : $\left\{ \begin{array}{l} \text{no further constraint will be set on} \\ \text{outputs of } S_1 \text{ other than the ones listed} \end{array} \right.$

This addendum states that S_1 has full control over its outputs and no other module can control them directly. Note the asymmetry between the handling of inputs and outputs: by exposing conditions for use, a module can set constraints on its environment. Not declaring a classification of ports into inputs and outputs allows for a bidirectional propagation of constraints between the module and its environment.

1) *Abstraction*: Abstracting amounts to relaxing constraints. An example of abstraction of S_1 is:

$$S_1^{(1)} : \left\{ \begin{array}{l} \text{ports: } x, y, z \\ \text{types: } x, y, z \in \mathbb{N} \\ \text{behaviors: } y \neq 0 \Rightarrow z = x/y \end{array} \right.$$

Ports were not changed. The input/output classification was removed, thus removing the addendum to S_1 . Types were relaxed (the mention of distance, time and speed was erased). Behaviors were relaxed since $[z = x/y] \vee \neg[y \neq 0]$. If type inference is supported in which \mathbb{R} subsumes \mathbb{N} , then the following is an abstraction of $S_1^{(1)}$:

$$S_1^{(2)} : \left\{ \begin{array}{l} \text{ports: } x, y, z \\ \text{types: } x, y, z \in \mathbb{R} \\ \text{behaviors: } y \neq 0 \Rightarrow z = x/y \end{array} \right.$$

Finally, denoting by symbol T the boolean constant “true” to indicate no constraint on entities,

$$S_1^{(3)} : \left\{ \begin{array}{l} \text{ports: } x, y \\ \text{types: } x, y \in \mathbb{R} \\ \text{behaviors: } T \end{array} \right.$$

is an abstraction of $S_1^{(2)}$ in which fewer ports are exposed. One can thus consider that $S_1, S_1^{(1)}, S_1^{(2)}$, and finally $S_1^{(3)}$, are successive abstractions of a same sub-system. These abstractions may be used at different stages of system design. In particular, in a bottom-up verification process, abstractions may be used instead of detailed descriptions to simplify analysis: if a property P_1 (also stated as a constraint) holds on the abstraction (say $S_1^{(2)} \models P_1$), then it also holds on the detailed design ($S_1 \models P_1$). An example of an (untyped) property such

that $S_1^{(2)} \models P_1$ is the following Assume/Guarantee type of statement:

$$P_1 : \left\{ \begin{array}{l} \text{ports: } x, y, z \\ \text{types: } T \\ \text{behaviors: } y \neq 0 \Rightarrow z = x/y \end{array} \right.$$

2) *Compatibility*: Consider another module S_2 :

$$S_2 : \left\{ \begin{array}{l} \text{ports: } \left\{ \begin{array}{l} \text{inputs: } z \\ \text{outputs: } y \end{array} \right. \\ \text{types: } y, z \in \mathbb{R} \\ \text{behaviors: } y = 2z \end{array} \right.$$

which induces, by convention, the following additional constraint:

addendum to S_2 : $\left\{ \begin{array}{l} \text{no further constraint will be set on} \\ \text{outputs of } S_2 \text{ other than the ones listed} \end{array} \right.$

S_2 satisfies the following property:

$$P_2 : \left\{ \begin{array}{l} \text{ports: } x, y, z \\ \text{types: } T \\ \text{behaviors: } y = 2z \end{array} \right.$$

in which types as well as the distinction of inputs and outputs have been abstracted.

Composing modules is by taking the conjunction of all the listed constraints, field per field. Say that a set of modules is *compatible* if the constraints resulting from the composition are all satisfiable. $S_1^{(2)}$ and S_2 are compatible, since their composition, denoted $S_1^{(2)} \parallel S_2$, yields the following set of satisfiable constraints:

$$\begin{array}{l} \text{ports: } x, y, z \\ \text{types: } x, y, z \in \mathbb{R} \\ \text{behaviors: } [y \neq 0 \Rightarrow z = x/y] \wedge [y = 2z] \end{array}$$

Regarding behaviors, the constraint boils down to $y \neq 0 \Rightarrow y = 2x/y$, which is satisfiable. Observe also that

$$S_1^{(2)} \models P_1 \text{ and } S_2 \models P_2 \implies (S_1^{(2)} \parallel S_2) \models (P_1 \parallel P_2)$$

Observe that $S_1^{(1)}$ and S_2 are also compatible if type inference is supported, since the types in S_2 can unify with those of $S_1^{(1)}$. And so are $S_1^{(3)}$ and S_2 . Regarding verification, since $S_1^{(2)}$ is an abstraction of $S_1^{(1)}$ and since $S_1^{(1)}$ and S_2 are compatible,

$$(S_1^{(2)} \parallel S_2) \models (P_1 \parallel P_2) \implies (S_1^{(1)} \parallel S_2) \models (P_1 \parallel P_2)$$

In contrast, we claim that S_2 and S_1 are not compatible. To see this we review the different fields of the specifications:

- ports: inputs and output match together and fit the rule that outputs of one module can only be connected to inputs of another module;
- types: if type inference is supported, then types of ports of S_2 can unify with the corresponding types in S_1 ;
- behaviors: taking conjunction yields $y \neq 0 \wedge z = x/y \wedge y = 2z$, which reduces to $y \neq 0 \wedge y = 2x/y$ and is certainly satisfiable;
- addenda: here we have a problem. The addendum to S_2 states that extra constraints are forbidden on y . However,

S_1 states such a constraint, namely $y \neq 0$. The conjunction of the two is not satisfiable.

Since $S_1^{(2)}$ and S_2 are compatible, the constraints stated by $S_1^{(2)} \parallel S_2$ are satisfiable, expressing that there exists an environment that is compatible with $S_1^{(2)} \parallel S_2$ — note that $S_1^{(2)} \parallel S_2$ is an open system since it has x as its input. Compatibility extends immediately to any finite set of modules.

3) *The notion of refinement*: Remembering that our theory must encompass both bottom-up and top-down compositional design, the following remarks can be formulated.

Recall that abstraction is by relaxing constraints. Consequently, abstraction preserves compatibility. Compositional verification works well with abstraction: if $S_1^{(1)} \models P_1$ and $S_2 \models P_2$, and if $S_1^{(1)}$ is an abstraction of some module S_1 , then, provided that S_1 and S_2 are compatible, it follows that $(S_1 \parallel S_2) \models (P_1 \parallel P_2)$. Compositional property checking is useful for both bottom-up and top-down compositional design.

Focus now on independent implementability. The two specifications $S_1^{(2)}$ and S_2 are compatible and their composition satisfies the desired property $P_1 \parallel P_2$. Unfortunately, developing $S_1^{(2)}$ independently will fail: S_1 is a correct implementation of $S_1^{(2)}$ but S_1 and S_2 are not compatible. In other words, *independent implementability calls for handling compatibility and the converse of abstraction, jointly*. This joint handling is what we call *refinement*.

In the next section we further develop the concepts of the previous section by proposing a “meta-theory”.¹ By this we mean a wish list of concepts, operators, and relations associated with requirements.

B. The Meta-Theory of Contracts

To introduce our meta-theory, we start from a model that consists of a universe of possible *designs*, each denoted by the symbol M ,² and a universe of their abstractions, or *contracts*, each denoted by the symbol \mathcal{C} . Our meta-theory does not presume any particular modeling style, neither for designs nor for contracts: for instance, some frameworks may represent designs and contracts with sets of discrete time or even continuous time traces, some theories use logics, other theories use state-based models of various kinds, and so on. We do assume that some operator of parallel composition has been defined both for designs, denoted by

$$M_1 \times M_2,$$

and for contracts or abstractions, denoted by

$$\mathcal{C}_1 \otimes \mathcal{C}_2.$$

Due to compatibility issues, operator \times is partial, not total, i.e., there may be cases in which $M_1 \times M_2$ is not well defined, we call it *illegal*. Reasons for $M_1 \times M_2$ being illegal depend on the mathematical nature of the representation of designs M . It may be due to static errors such as type mismatch, interface

¹This term is of course reminiscent of “meta-model”; a meta-model states the rules a family of models should obey.

²The symbol M that we use to refer to designs is reminiscent of “Machine”.

mismatch, but also dynamic run-time errors such as deadlocks, or races. Again, this depends on the mathematical framework in consideration.

Note that it may very well be that $M_1 \times M_2$ is illegal, whereas $M_1 \times M_2 \times E$ is not, where E denotes some possible environment for $M_1 \times M_2$. An example of such a situation is when illegal compositions correspond to the existence of deadlocks because environment E can prevent $M_1 \times M_2$ from reaching the deadlock configuration.

While the mathematical nature of the representation of designs M and contracts \mathcal{C} depends on the particular framework considered, we can establish several additional generic operators or relations that can be constructed from the simple elements of the meta-theory. An overview of these operators is shown in Table II. Our starting point is a binary relation between designs and contracts, denoted by the symbol \models , which we assume is defined in the model, and from which we derive the other notions in a simple and direct way. The next sections describes these elements in detail.

1) *Implementation*: We say that design M *implements* contract \mathcal{C} , written

$$M \models \mathcal{C} \quad (1)$$

if M is correct with respect to the considered contract \mathcal{C} in the intuitive sense of Property 2 of Section II. The mathematical definition for this implementation relation depends on the particular framework for contracts. Sometimes this relation takes different meanings, depending on the context, but can generally be identified with one of the following:

$$M \models \mathcal{C} : \begin{cases} M \text{ satisfies } \mathcal{C}, \text{ or} \\ M \text{ abstracts as } \mathcal{C}, \text{ or} \\ M \text{ implements } \mathcal{C}. \end{cases}$$

Throughout this section, implementation relation (1) is considered as given and should be taken as an “axiom”. We derive all other constructions of the meta-theory from this given notion.

2) *Contracts*: Building on the implementation relation (1), the semantics of a contract \mathcal{C} is the characteristic set $\chi_{\mathcal{C}}$ of all its correct implementations:

$$\chi_{\mathcal{C}} \triangleq \{M \mid M \models \mathcal{C}\} \quad (2)$$

For reasons of effectiveness, in most contract theories, contracts are finitely described. As a consequence, it is not true that every set χ of designs can be represented as $\chi = \chi_{\mathcal{C}}$ for some contract \mathcal{C} .

3) *Refinement*: \mathcal{C}' *refines* \mathcal{C} if any implementation of \mathcal{C}' is an implementation of \mathcal{C} :

$$\mathcal{C}' \leq \mathcal{C} \text{ if and only if } \chi_{\mathcal{C}'} \subseteq \chi_{\mathcal{C}} \quad (3)$$

4) *Conjunction*: A design M implements the *conjunction* of contracts \mathcal{C}_1 and \mathcal{C}_2 if and only if it implements both contracts. The intent is thus to define the conjunction of contracts as being the contract with characteristic set $\chi_{\mathcal{C}_1} \cap \chi_{\mathcal{C}_2}$. However, not every set of designs is the characteristic set of a

Operator or Property	Definition	Comment
Design	M is either <i>legal</i> or <i>illegal</i>	Depends on the particular theory of contracts
(1) Implementation	$M \models C$	Depends on the particular theory of contracts
(2) Contract	The semantics of C is $\chi_C = \{M \mid M \models C\}$	C is characterized by the set of its implementations
(3) Refinement	$C' \leq C$ iff $\forall M : M \models C' \Rightarrow M \models C$	Every implementation of C' is an implementation of C
(4) Conjunction	$C_1 \wedge C_2 = \max\{C \mid C \leq C_1 \text{ and } C \leq C_2\}$	Greatest lower bound for refinement
(5) Product	$C_1 \otimes C_2 = \min \left\{ C \mid \begin{array}{l} M_1 \models C_1 \text{ and } M_2 \models C_2 \\ \Downarrow \\ M_1 \times M_2 \models C \end{array} \right\}$	Builds on top of parallel composition for designs
(6) Consistency	C is <i>consistent</i> iff $\chi_C \neq \emptyset$	There exist correct designs for C
(7) Compatibility	M is <i>compatible</i> iff $\exists E : M \times E$ legal C is <i>compatible</i> iff $\exists E : \forall M \models C \Rightarrow M \times E$ legal	M possesses an environment E such that $M \times E$ is legal C possesses an E compatible with any implementation of C
(8) Quotient	$C_1 / C_2 = \max\{C \mid C \otimes C_2 \leq C_1\}$	Specifies how to patch C_2 in order to best approximate C_1

Table II
Summary of the meta-theory of contracts; (1) indicates that this line is discussed when introducing equation (1).

contract. The best approximation consists in taking the greatest lower bound for the refinement relation:

$$C_1 \wedge C_2 \equiv \max\{C \mid C \leq C_1 \text{ and } C \leq C_2\} \quad (4)$$

The conjunction of contracts formalizes what it means to have several requirements in a requirement document. It also formalizes what it means to structure requirement documents into several viewpoints, for a system or a component.

5) *Product*: Here we focus on the hierarchical structure of contracts of the system and its sub-systems. Using the composition operation \times for sub-systems, the intent is to define the product of contracts attached to two different sub-systems as being the contract with characteristic set $\{M_1 \times M_2 \mid M_1 \models C_1 \text{ and } M_2 \models C_2\}$. The rationale here is that a design implements the composition $C_1 \otimes C_2$ if and only if it is the \times -product of two designs that implement C_1 and C_2 , respectively. Again, no such contract exists in general and thus we take the best approximation of that characteristic set:

$$C_1 \otimes C_2 \equiv \min \left\{ C \mid \begin{array}{l} M_1 \models C_1 \text{ and } M_2 \models C_2 \\ \Downarrow \\ M_1 \times M_2 \models C \end{array} \right\} \quad (5)$$

The \otimes -product of contracts prepares the support for separate development of sub-systems. It is not sufficient, however, since the issue of compatibility is not handled at this point: the definition of the product does not refer to $M_1 \times M_2$ being legal or illegal. Compatibility is addressed separately.

At this point, we must discuss algebraic properties that operations \times and \otimes must have in order to support the needed flexibility in system design. In complex systems, the functional high-level architecture may not match the actual deployed architecture for the implementation. Re-architecturing is typical when performing deployment, see Figure 2. In this figure, we illustrate the mapping of three sub-systems to a distributed architecture consisting of two computing units and one bus. Components M_1, M_3 , and M_4 are mapped to the first computing unit, whereas components M_2, M_5 , and

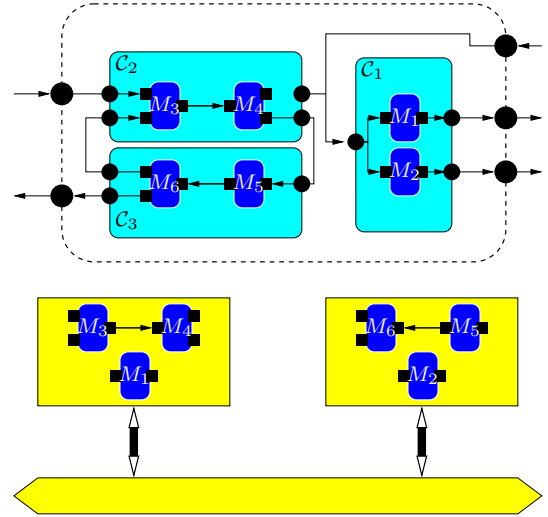


Figure 2. Re-architecturing. Top: application architecture. Bottom: deployed computing architecture.

M_6 are mapped to the second one. Wires of the functional architecture are mapped to frames of the bus accordingly — this is not shown. To preserve the semantics, such a re-architecturing requires that the composition operator \times for designs is *associative*: $(M_1 \times M_2) \times (M_3 \times M_4) \times (M_5 \times M_6) = (M_1 \times M_3 \times M_4) \times (M_2 \times M_5 \times M_6)$. Commutativity is also clearly requested. Commutativity and associativity of \times is thus a requirement on the model used to represent designs. By (5), contract composition \otimes inherits associativity and commutativity.

6) *Consistency*: In the intuitive setting we discussed in part 1, Section II-E [16], a contract is called consistent if its different requirements “do not contradict each other”. Equivalently, using characterization (2) for contracts, a contract is consistent if it is feasible to develop implementations for it:

$$C \text{ is consistent iff } \chi_C \neq \emptyset \quad (6)$$

7) *Compatibility*: We have distinguished illegal from legal designs. How does this notion lift to contracts? We could define as illegal a contract C possessing at least one illegal implementation. This is too pessimistic an approach, however. Since we are interested in open, not closed, systems, we are interested in designs M possessing at least one environment E such that $M \times E$ is legal — M itself can still be illegal. Call such M *compatible* and lift this definition to contracts:

$$\begin{aligned} M \text{ is compatible} &\text{ iff } \exists E : M \times E \text{ is legal} \\ C \text{ is compatible} &\text{ iff } \exists E : \forall M \models C \Rightarrow M \times E \text{ is legal} \end{aligned} \quad (7)$$

The usage is to associate the term “compatible” to pairs of designs or contracts. Our definition encompasses this usage by applying (7) to $M_1 \times M_2$ or $C_1 \otimes C_2$.

8) *Quotient*: The adjoint of the product operation \otimes is also a useful operator:

$$C_1 / C_2 = \max\{C \mid C \otimes C_2 \leq C_1\} \quad (8)$$

C_1 / C_2 is the most general context C in which C_2 refines C_1 . It formalizes the practice of “patching” a design to make it behaving according to another specification.

C. Discussion

Of course, the definitions in Table II are not effective. But they can serve as a reference for the corresponding definitions stated in each given framework. Some frameworks instantiate only part of the meta-theory, leaving aside some of the notions (typically, conjunction, compatibility, and/or quotient).

How many sets of designs can be represented as the characteristic set of some contract depends on the theory — this is a measure of the expressiveness of the theory. Expressiveness comes at the price of higher complexity.

One important issue is that of compatibility. While consistency is a satisfiability problem, compatibility is much more involved. Powerful theories provide means for a contract to explicitly expose which environment is compatible for it.

In sections IV–VI we develop three frameworks that we believe are the most interesting. For each of them we analyze how far they fulfill the meta-theory. Then, we provide a broader review where we also briefly discuss other candidate frameworks and contrast them to the meta-theory, too.

Although MDE belongs to “related work” rather than a candidate contract theory, we briefly discuss it here because it has historical importance and benefits from wider awareness than the more novel theories developed in this paper. The two concepts of interface and contract first emerged in the area of *Object Oriented programming* [105], [1], [116]. They were subsequently developed to maturity by giving rise to MDE (*Model Driven Engineering*) [86], [119], [96]. 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) [107], [125].³ So far nothing is said in system architectures regarding functional and extra-functional properties of interfaces (behaviors and performance). To account for behavior and performance, the classical approach in MDE then consists in enriching components with *methods* that can be invoked from outside, and/or *state machines*. Attributes on port methods have been used to represent non-functional requirements or provisions of a component [33]. 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 [12], [97], [112]. The notion of refinement is replaced by the concept of class inheritance. Inheritance, however, is unable to cover aspects related to behavior refinement. Nor is it made precise what it means to take the conjunction of interfaces, only approximated by multiple inheritance, or to compose them. This is way behind the meta-theory of Table II.

D. Casting vertical contracts in the meta-theory

We complete this section by casting into our meta-theory the concept of *vertical contract* introduced in companion paper [16]. In companion paper [16], the so-called vertical contracts are emphasized in the context of layered design, virtual prototyping, and particularly in platform-based design (PBD).

Let us recall this concept using Figure 2. This figure illustrates the deployment of an application architecture (shown in blue) over an execution platform consisting of two ECUs and a bus (shown in yellow). When dealing with deployment, besides the functional viewpoint, the two safety and timing viewpoints are of importance. The safety viewpoint is concerned with the risk of primary faults and failures, how they could propagate throughout the system, and what the consequences for the resilience of the overall system would be in terms of failure probability. The timing viewpoint collects timing requirements regarding the application (requirements on latencies, throughput, jitter, and schedulability).

The effective satisfaction of safety or timing viewpoints by a considered deployment depends on 1/ the supporting execution platform and 2/ the mapping of the application to this execution platform. This is why we call *vertical* a contract stating, for an application architecture, obligations regarding end-to-end timing or system wide safety, based on corresponding assumptions on the execution platform. Vertical contracts are an intuitive concept and play an important methodological role. Since we do not offer a mathematical notion of vertical contract, we must explain how this notion can be cast in our meta-theory. We do this next in two ways

³Roughly 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.

— in the coming explanations, we write in italics the concepts of the meta-theory we refer to.

Vertical contracts through refinement: In the first and most direct explanation, checking that the whole deployed architecture (Figure 2, bottom) satisfies the vertical contract stated on the whole application architecture (Figure 2, top) amounts to checking whether the former is a *refinement*, or even an *implementation* of the latter. Such refinement checking can be performed on the overall system — this, however, does not scale up to complex systems. By using the property (5) of independent development, we can check refinement on each sub-system: $\mathcal{C}'_1 \leq \mathcal{C}_1$ and $\mathcal{C}'_2 \leq \mathcal{C}_2$ imply $(\mathcal{C}'_1 \otimes \mathcal{C}'_2) \leq (\mathcal{C}_1 \otimes \mathcal{C}_2)$, thus making the handling of vertical assumptions through refinement seemingly modular. This is not quite true in general, however, since the above modular reasoning only works if the execution architecture matches the application architecture, see Figure 2 for a frequently encountered situation where this does not occur. Thus, while being interesting, this way of formalizing vertical contracts is not sufficient and we propose another approach that better supports modularity, even in the context of Figure 2.

Vertical contracts by mapping application to execution platform: The coming explanation is more interesting as it relates to virtual engineering. Suppose one has a virtual model of the execution platform as an architecture composed of several components. Such components would, in the context of Figure 2, consist of a description of the available computing units, a description of the bus protocol and frame structure, and/or a library of RTOS (Real-Time Operating System) services. The resulting components are enhanced with timing information (for the timing viewpoint) or MTBF (Mean Time Between Failures) and fault propagation information (for the safety viewpoint). The mapping of the application to the execution platform is by taking the *parallel composition* of the application architecture and the execution platform as follows. Let

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

be the two contracts describing the application and execution platform, respectively. Contract \mathcal{C} for the application is the conjunction \bigwedge of different viewpoints ranging over index k , which are defined compositionally — here we assume that the different viewpoints are developed for the whole application, and then combined by conjunction. Contract \mathcal{P} for the platform is the parallel composition \bigotimes of the different components ranging over j , whose contract is the conjunction of the same viewpoints — here we consider that, in the library modeling the execution platform, each component comes with all its viewpoints.⁴

The mapping of the application over the architecture is modeled by the parallel composition $\mathcal{C} \otimes \mathcal{P}$ defined by the following set of *synchronization tuples* — recall that, in a synchronization tuple, both occurrences and values of the

different elements are unified.⁵ This is illustrated in Figure 3.

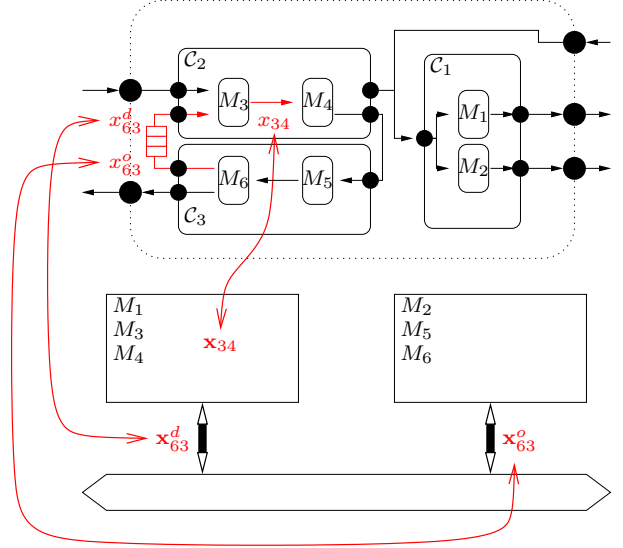


Figure 3. Mapping the application to the execution platform as $\mathcal{C} \otimes \mathcal{P}$.

In this figure, synchronization pairs are depicted in red.

- The wire linking M_3 to M_4 in the application (we denote it by x_{34}) is synchronized, in the deployment, with the output of the module implementing M_3 and the input of the module implementing M_4 in the left computing unit. And similarly for the wire x_{56} .
- In the deployment, the wire x_{63} linking M_6 to M_3 must traverse the bus. This is captured in the following way. We first refine the application architecture by distinguishing, in the application architecture, the origin of this wire from its destination: call them x_{63}^o and x_{63}^d , respectively. In the refined application architecture, we assume that x_{63}^o and x_{63}^d are related by some buffer of bounded size. The deployment is then captured by the following two synchronization pairs. The first one synchronizes x_{63}^o with the output of the module implementing M_6 in the right computing unit. The second one synchronizes x_{63}^d with the input of the module implementing M_3 in the left computing unit.

With this formalization of the deployment as the parallel composition

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

it remains to check whether $\mathcal{C} \otimes \mathcal{P}$ refines \mathcal{C} , which can then be performed compositionally.

This technique of mapping application to its execution platform is used in RT-Builder⁶ and Metropolis [11], [59].

E. Conclusions

We have developed a meta-theory that expresses the properties that a contract theory should satisfy in order to consistently

⁴These are just methodological proposals, we could have done differently.

⁵To simplify, we follow a data-flow interpretation of the different diagrams.

⁶<http://www.geensoft.com/en/article/rtbuilder>

Example 1 (test for Requirement 2.2 in Scade):

```
node test42(entry_gate_open,
           parking_full: bool)
  returns(out: bool)
let
  out = not entry_gate_open or
        not parking_full
tel
```

Example 2 (test for Requirement 2.2 in PSL):

```
never(parking_full && entry_gate_open)
```

Example 3 (test for Requirement 2.2 as an automaton):

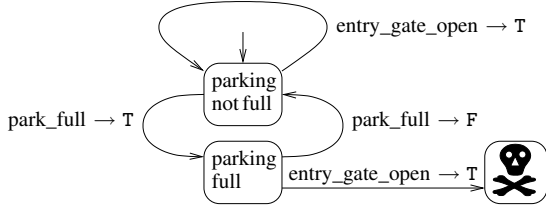


Figure 4. Test for Requirement 2.2 in Section II: in Scade, in PSL, and as an automaton where the trap state indicates violation of the property.

represent the design intent across several phases of the design methodology, including the collection of requirements, their organization into viewpoints, and their decomposition into components. In the next sections we will review instantiations of the meta-theory, and discuss their degree of coverage of the capabilities listed in Table II. This review does not cover process algebraic frameworks mixing parallel product and conjunction [80], [100] which in principle address the concepts 4 and 5 of the meta-theory. We do this because in these approaches processes are defined over a universal alphabet, hence missing the issue of locality of alphabets. More importantly, no attention is paid to the role of the environment versus the system.

IV. OBSERVERS AS CONTRACTS

Observers, also called monitors, are commonly used in design as a way of verifying whether a design satisfies a property. Observers monitors variables of the design as they evolve to check whether they are outside a specified range. They are associated to tests, a way to represent properties.

A. Tests

Tests are simple ways to deal with contracts:

Definition 1 (test): A test is a program attached to a property (or requirement), satisfying the following conditions. It is deterministic and takes all ports of the considered interface as inputs and delivers a boolean output. This boolean output switches from its initial status *true* to the status *false* as soon as the considered property or requirement gets falsified and then remains false for ever.⁷

Indeed, in many industrial verification flows, tests are attached to requirements. For the case of Requirement 2.2 in Section II,

⁷The terms OK and KO used in Figure 5 correspond to *true* and *false*, respectively.

which states that “the entry gate may open only if the parking is not full”, such a test can be expressed in different ways, see Figure 4. One possibility is to use Scade⁸, where Requirement 2.2 is violated if *out* gets the value “true”. Equivalently, this test can be expressed using the PSL language [108] with the same naming conventions as for the Scade test. Finally, we provide a state machine description for the same test. In fact, any executable language can be used to express tests. Languages with formal semantics are preferred, however, as they offer reproducible executions and support formal analysis better — Figure 4 depicts such examples. Using tests for checking properties [122], [98], [84], [31], [35] is illustrated on Figure 5.

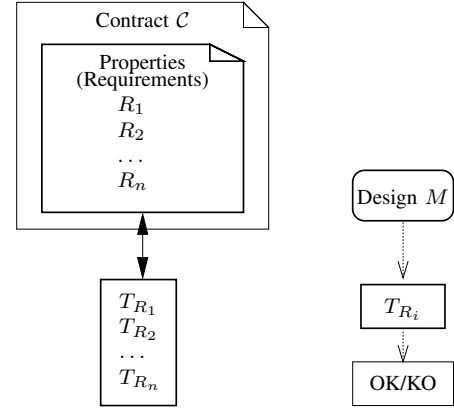


Figure 5. Test based interfaces, naive form.

B. Observers

Observers [126], [81] are built on top of tests and provide the simplest formalism to represent contracts, by explicitly considering the context or environment in which the considered component is supposed to operate. This leads to specify properties for both the component (the *guarantees*) and its environment (the *assumptions*), collected as the *contract* for the component. This is illustrated on Figure 6 under the name of *observer*. Observers are attached to a contract partitioned into its assumptions and guarantees $\mathcal{C} = (A_1 \dots A_n, G_1 \dots G_m)$. Observers are intended to monitor the violation of any of the guarantees, subject to the satisfaction of the assumptions. The corresponding definition is given now:

Definition 2: An observer is a tuple $O = (T_{A_1} \dots T_{A_n}, T_G)$ of tests associated to requirements $A_1 \dots A_n$ and G , where G denotes any of the guarantees $G_1 \dots G_m$.

Tests T_{A_i} are used to filter out histories from the design that do not conform to the assumptions. For those remaining legal histories, test T_G is used to check the satisfaction of the obligation G by the design. In our example of Requirements 2 for the parking lot, one observer is attached to every obligation (Requirement 2.2, 2.3, etc.). Each of these observers must only be fed with port histories that are *legal*, i.e., satisfy all

⁸<http://www.esterel-technologies.com/products/scade-suite/>

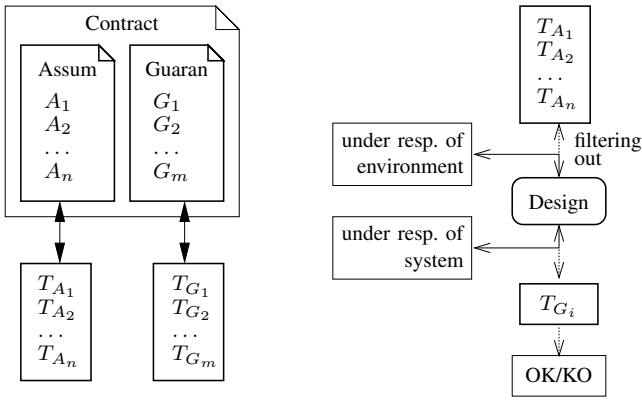


Figure 6. Observer based interfaces

assumptions (the conjunction of Requirements 2.1, 2.7 and 2.11). By doing so, each individual obligation of the parking lot is monitored for possible violation. This is illustrated on Figure 6.

Note the asymmetry in handling assumptions and guarantees. In particular, adding one more assumption impacts the entire generation of legal inputs, an inconvenient lack of modularity. For the same reason, no modular use of observers exists for contracts structured into chapters or viewpoints.

C. Observers as a Theory of Contracts

Let us now compare observers to the meta-theory of Table II. The way the implementation relation $M \models C$ is performed using testing is depicted on Figure 6. As for testing, this only provides a semi-decision procedure in that failure to satisfy a contract can be detected in this way, but satisfaction cannot be proved except when exhaustive testing is feasible.

Operators or relations (2)–(6) of Table II follow from implementation relation (1), as byproducts. As for testing, it is not possible to compute them using observers only. How closely can then we approach them?

Focus on refinement relation (3). Can we disprove refinement using observers? Yes, as it is sufficient to exhibit *some* design for which one of the tests associated to guarantees returns *false*, whereas all tests corresponding to assumptions return *true*. Proving refinement is out of reach even if exhaustive testing of designs is feasible, since one has to prove that *every* design passing the observers for C' also passes the observers for C . In contrast to testing, *the conjunction operator (4) cannot be represented in terms of observers*, due to the monolithic handling of assumptions, see above.

Focus now on parallel composition (5). Let

$$(T_{A_{1,1}} \dots T_{A_{1,n_1}}, T_{G_{1,1}} \dots T_{G_{1,m_1}}) \text{ and} \\ (T_{A_{2,1}} \dots T_{A_{2,n_2}}, T_{G_{2,1}} \dots T_{G_{2,m_2}})$$

be the observers associated to contracts C_1 and C_2 , respectively.

Consider the following condition:⁹

ports or variables involved in the interaction between M_1 and M_2 in design $M = M_1 \times M_2$ are not hidden. (9)

If (9) holds, then we can evaluate the two observers on M . Failure to pass for some observer indicates failure of M to satisfy contract $C_1 \otimes C_2$. Thus, provided that ports or variables involved in the interaction between M_1 and M_2 in design $M = M_1 \times M_2$ are not hidden, contract $C_1 \otimes C_2$ is represented by joining the two observers. If, however, (9) does not hold, then contract $C_1 \otimes C_2$ cannot be represented.

Finally, the two notions of compatibility (7) and quotient (8) of Table II are not addressed by observers.

By allowing for on-line monitoring of contracts, observers raise questions regarding the notion (1) of implementation or correctness in Table II. The issue arises when both the assumptions and guarantees can get violated but at different times. Figure 7 depicts a case where guarantees get violated first and then assumptions get violated, too. One may question

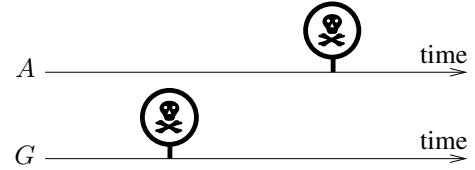


Figure 7. Violating guarantees and then assumptions yields contract violation (violation is indicated by the friendly pictogram).

whether the referred contract should be considered as being again satisfied after the assumptions get violated. Contract based approaches adopt the convention that, once the implication $A \Rightarrow G$ gets violated, then the contract is considered violated forever and this fact is not defeasible. Thus, the case of Figure 7 yields a contract violation. Of course, if the converse order occurs (assumptions get violated first), then the contract is met since the implication $A \Rightarrow G$ is not falsified.

We now detail how the approach based on observers can be implemented using actual languages to express tests. We review the use of synchronous languages (Scade, Signal, and to some extent Simulink), the Property Specification Language (PSL), and Live Sequence Charts and Play Engines developed by David Harel. For each of them we sketch how tests are associated to requirements or properties and we point to the available technology and tools.

D. Implementing observers

1) *Using Simulink/Scade/Signal*: Synchronous languages [17], [73], [19] are a formalism of choice in dealing with observers. The family of Synchronous Languages comprises mainly the imperative language Esterel [68], [62] and the

⁹ Condition (9) is reasonable if the overall design M decomposes as $M = M_1 \times M_2$ by following the *same* architecture as for the specification $C = C_1 \otimes C_2$ — in other words, design architecture meets specification architecture. Unfortunately, this is typically not the case for complex systems, see Figures 2 and 3.

- *Property simulation*: although PSL is a declarative and non-executable language, the tool proposes a method for automatically extracting histories (behaviors) that either satisfy or violate a property. The designer has the choice to further explore her properties by adding additional constraints on the behaviors that the tool generates.
- *Property assurance*: given a contract consisting of a set of requirements, the designer can check the consistency of her contract. She can in addition define properties that are used to cross-check the contract. In particular, the designer can query the tool whether her contract satisfies these additional properties. Finally, the designer can check whether her contract offers a possibility for some scenario to happen, where the scenario is specified as another PSL property.

As previously stated, PSL is built on top of LTL and regular expressions. It follows that one can express liveness properties in PSL, that 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 [65]. A survey of using PSL in runtime verification can be found in [63].

Like many other observer-based specification languages, PSL does not support modalities *may* and *must* such as occurring in Requirements 1.1 and 1.3. Thus a translation would ignore them, typically by interpreting the *may* and the *must* in the same way; this is an important limitation regarding requirement 1.3. Also, PSL is not expressive enough to specify probabilistic properties needed by the reliability documents expressing safety properties (Requirements 5). Finally, the quantitative relation between the cost of parking and the time spent would require a finite domain approximation of the values and would result in a cumbersome notation (Requirement 1.4).

3) *Using LSCs and Play Engines*: The following brief introduction is borrowed from [77]. *Live Sequence Charts* (LSC) [44], [79] is a graphical specification language based on scenarios that is an extension of Message Sequence Charts (MSC)¹⁵. Both contain vertical lines, termed lifelines, which denote objects, and events that involve one or more of these objects. The most basic construct of the language are messages: a message is denoted by an arrow between two lifelines (or from a lifeline to itself), representing the event of the source object sending a message to the target object. More advanced constructs, like conditions, if-then-else, loops, etc., can also be expressed. A typical LSC consists of a prechart (denoted by a blue dashed hexagon), and a main chart (denoted by a solid frame). Roughly, the intended semantics is that whenever the prechart is satisfied in a run of the system, eventually the main chart must also be satisfied (see Figure 9). LSCs are multi-modal; almost any construct in the language can be either *cold* (usually denoted by the color blue) or *hot* (denoted by red), with a semantics of “may happen” or “must

happen”, respectively. If a cold element is violated (say a condition that is not true when reached), this is considered a legal behavior and some appropriate action is taken. Violation of a hot element, however, is considered a violation of the specification and is not allowed to happen in an execution.

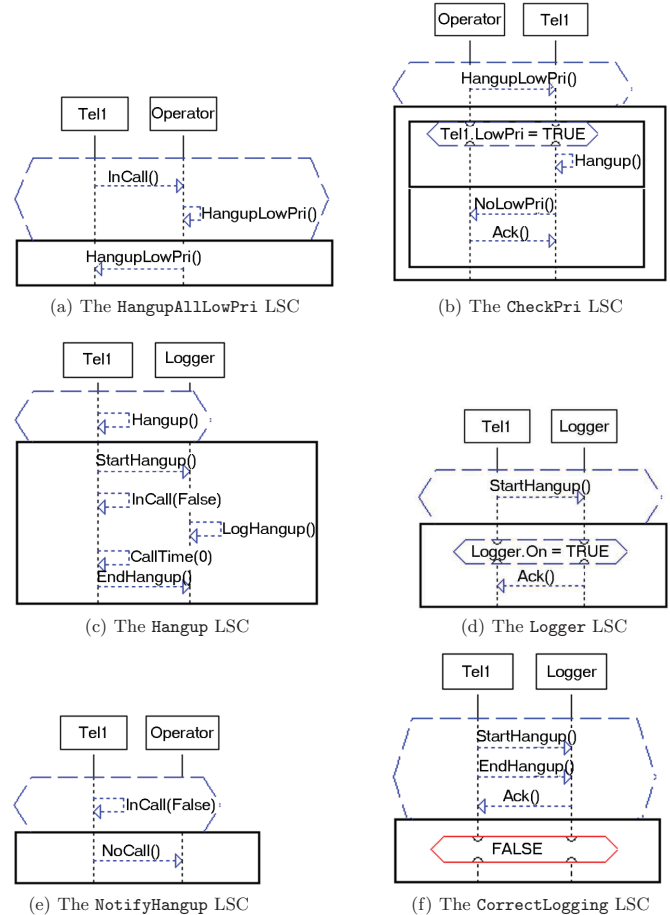


Figure 9. An LSC specification with six charts for a simple phone system in which the operator may decide to hang up low priority calls. Borrowed from [77].

Figure 9 shows an example of LSC specification. It describes a simple telephone system with three objects — a phone, an operator, and a logger. The operator may wish to force low-priority calls to disconnect (e.g., due to a system overload). We do not discuss here all charts, but only discuss some features of interest for us. The interested reader is referred to [77] for a complete explanation. The graphical style of LSC makes it almost self-explanatory. For example, the LSC *HangupAllLowPri*, in chart (a), refers to the operator notifying the phone that a low-priority hangup is called for. Specifically, the LSC states that if the phone sends the message *InCall* to the operator, and the operator sends *HangupLowPri* to itself, then the operator sends *HangupLowPri* to the phone. Everything is cold. As in most scenario-based languages, events are ordered by causality. Causality follows either from sitting in a same time line,

¹⁵<http://www.sdl-forum.org/MSC/index.htm>

or is due to the circulation of messages — sending causes reception. The prechart precedes the main chart. Chart (c) shows an example in which events are not all ordered in the main chart. The LSC `Logger`, in chart (d), specifies how the logger replies to the telephone: If the phone sends `StartHangup` to the logger, the `Logger:On` condition is checked. If it is true, the execution continues, and the logger sends `Ack` to the phone. If the `Logger:On` is false, then being a cold condition, the chart exits gracefully and the `Ack` is not sent. Finally, the LSC `CorrectLogging`, given in chart (f), is an anti-scenario; it states that the scenario in which the three messages, `StartHangup` from the phone to the logger, `EndHangup` from the phone to the logger, and `Ack` from the logger to the phone, are sent in this order is forbidden. This illustrates the use of negative-hot status specified by the hot condition “false”.

LSC are executable. The execution of a set of LSCs is called *play out* by the authors [79]. Following [77], executions progress from one chart to another in the following way. Play-out remembers at each point in time the set of active LSCs (those for which the prechart has already completed, but the main chart has not), and for each such LSC it holds the current cut (listing what has already happened, and what has not). At each step, the play-out mechanism chooses one message that is enabled in some LSC (i.e., it appears directly after the current cut), and does not violate any other chart (a message is violating if it appears in an active chart but is not enabled in it), and executes it. Non-determinism that can occur (with the possibility of executing several charts) is solved on-the-fly. More clever look-ahead policies for executing charts are reviewed in [77]. As a single thing to remember, the reader should understand that precharts are sort of “flexible” states, from which actions in the form of main scenarios can be performed — precharts should not be confused with assumptions, however. Play out is useful for checking an LSC specification against a property [46].

Also, LSC can be used as (expressive) tests to monitor the execution of a design over non-terminating runs. While doing so, the same rules as above are applied, except that the design controls the emission of events and the LSC specification is used to react in case of violation of the specification. This is manifested by the absence of rules explaining the occurrence of some event at run-time. This procedure is implemented by the Tracer tool¹⁶ of David Harel’s team.

From the preceding discussion, LSCs are very well suited to express contracts in the form of observers. Regarding LSC and its use in our context, the up-to-date development by the team of David Harel is PlayGo [78]. PlayGo is a comprehensive tool for scenario-based programming, built around the language of Live Sequence Charts and its Java derivative, and the Play-in/Play-out approach. The interested reader is referred to the PlayGo web site¹⁷ for further information.

More developments have been performed by Damm and

co-workers about LSC in the context of observers. In [87], Klose introduces LSC specifications to complement the model-based development of intra-object behaviour of a systems’ specification. The verification then may use an observer driven approach, where the system model is composed with an observer automaton [87], or a temporal logic formula [45], both derived from the original LSC specification. A graphical editor, the LSCedit [87], is available in order to draw and manipulate LSC specifications. Whether an intra-object description, for example expressed in terms of Statemate state-charts or UML state-machines, adheres to an LSC specification can automatically be established by model-checking [28], [87], [118]. In [47], the verification of UML models with respect to LSC specifications is extended to the case of unbounded object creation. Here, the structure of the LSC specification is exploited to obtain a finite number of representative queries expressed as temporal logic formulae. Also, the UML model itself is abstracted to a finite representation. This approach has been practically implemented and demonstrated in [127]. Recently, the observer driven approach was extended to be used in the real-time setting with the Uppaal model checker as back-end.

E. Discussion

To summarize, *observers* are a first beneficial step toward introducing contracts. By distinguishing between the actions of the environment and those of the component, observers support the distinction between assumptions and guarantees. By using observers, it is already possible to say something useful regarding the conformance of a design with respect to a contract. Under the restriction (9), parallel composition of contracts can be handled in part. Conjunction cannot be captured, however. In addition, unless testing can be done exhaustively, using observers cannot provide a sound and complete instantiation of the different concepts of Table II. On the other hand, since observers only require simulating a design, they support any kind of property or data type — data types need not be finite, quantitative reasoning can cover real-time, continuous time dynamics (as in the example of analog circuit design of companion paper [16]), and so on. Thus, observer based contracts have their advantages. They constitute the first, easy, step toward using contracts.

V. ASSUME/GUARANTEE CONTRACTS

So far we did not answer any of the questions raised in Section II on the parking garage example, see Property 3 regarding refinement and Question 1 on the conjunction of viewpoints and the subsequent discussion. In this section we will be able to answer these questions for the first time.

There are several variants of the concept we develop in this section. In all cases, assumptions A and guarantees G will be *assertions*, i.e., sets of behaviors of actions, signals, and/or variables involved in a design. Accordingly, operations on assumptions and guarantees will be expressed using either notations from logic (\wedge, \vee, \neg) or corresponding set theoretic

¹⁶<http://www.wisdom.weizmann.ac.il/~maozs/tracer/>

¹⁷http://www.wisdom.weizmann.ac.il/mediawiki/playgo/index.php/Main_Page

notations $(\cap, \cup, (\cdot)^c)$, according to convenience. The implication $A \Rightarrow B$ is defined as either $B \vee \neg A$ or $B \cup A^c$. Similarly, designs M and environments E are represented by their sets of all possible behaviors of actions, signals, and/or variables involved in them, i.e., as an assertion. Operations on designs and environments are thus set theoretic. Designs or environments come equipped with a notion of *composition* denoted by $M \times N$ and defined as the intersection $M \cap N$ of the corresponding assertions. Prior to developing our preferred variant, we review a few alternatives.

In its basic variant, an Assume/Guarantee contract is simply a pair $\mathcal{C} = (A, G)$ of assertions. For M a design and \mathcal{C} a contract, say that $M \models \mathcal{C}$ if M satisfies the implication $A \Rightarrow G$, that is, M satisfies $G \vee \neg A$. Consequently, if M is operating under an environment E satisfying A , then G is guaranteed. Since $M \models \mathcal{C}$ if and only if M satisfies $G \vee \neg A$, the first idea when considering Assume/Guarantee contracts is to identify contract $\mathcal{C} = (A, G)$ with the property characterizing its implementations, namely $P_{\mathcal{C}} = G \vee \neg A$. With this characterization, \mathcal{C}' refines \mathcal{C} if and only if $P_{\mathcal{C}'} \Rightarrow P_{\mathcal{C}}$. Observe that different pairs (A, G) of assumptions and guarantees can lead to the same contract, since only property $G \vee \neg A$ matters. In particular, knowing that (A', G') refines (A, G) does not set constraints on A' and G' individually. Also, $\mathcal{C}_1 \wedge \mathcal{C}_2$ has no “natural” expression as a pair of assumptions and guarantees and the same holds for parallel composition $\mathcal{C}_1 \otimes \mathcal{C}_2$. Finally, compatibility is not at all addressed. Note that the failure to manipulate assumptions and guarantees explicitly mirrors the difficulties we had with observers. To cope with this non satisfactory situation, authors have proposed various work-arounds.

While keeping the definition of implementation relation unchanged ($M \models \mathcal{C}$ if and only if M satisfies $G \vee \neg A$), researchers from the SPEEDS project [18] have proposed to reinforce the refinement relation as follows: say that $\mathcal{C}' \leq \mathcal{C}$ if $A' \Leftarrow A$ and $G' \Rightarrow G$, meaning that the refined contract offers a stronger guarantee under a more relaxed assumption. Observe that, when defined in this way, $\mathcal{C}' \leq \mathcal{C}$ implies $P_{\mathcal{C}'} \Rightarrow P_{\mathcal{C}}$ but the converse is not true. Said differently, refinement is sound but not complete and this definition for refinement approximates definition (3) of Table II. The effective formula corresponding to the conjunction (4) of Table II is a direct consequence of the definition of refinement, since the conjunction must be the least upper bound operator associated with the refinement order:

$$(A_1, G_1) \wedge (A_2, G_2) = (A_1 \vee A_2, G_1 \wedge G_2).$$

Some care is needed for the formula defining parallel composition in the SPEEDS setting. Recall that different contracts may possess identical sets of implementations, since the latter is characterized by the formula $G \vee \neg A$. A contract $\mathcal{C} = (A, G)$ is said to be in *canonical form* if $G \vee \neg A = G$; observe that replacing (A, G) by $(A, G \vee \neg A)$ yields an equivalent contract that is in canonical form. Then, for \mathcal{C}_1 and \mathcal{C}_2 in canonical form, their parallel composition (corresponding to

(5) in Table II) is defined as

$$\mathcal{C}_1 \otimes \mathcal{C}_2 = ([(A_1 \wedge A_2) \vee \neg(G_1 \wedge G_2)], [G_1 \wedge G_2]).$$

The rationale for such a formula for the assumption is that behaviors which are guaranteed not to occur by either of the components can be safely assigned to the environment. At the same time, the formula takes care of discharging those assumptions which are already satisfied by the other component, and which are longer under the control of environment.

Inspired by [91], another form for Assume/Guarantee contracts was proposed by [71], [72] when designs are expressed using the BIP programming language [25], [121]. 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.

In the following subsections, we develop our preferred variant, which builds on the original work of the SPEEDS project [18] and is a reformulation of a proposal by Damm and Hungar [43].

A. Assume/Guarantee contracts with strong/weak assumptions

In this framework, assumptions are categorized into *weak* and *strong* ones. Weak assumptions are handled as implications. Thus, when operating in an environment not meeting the weak assumptions, the considered component may not guarantee anything. Strong assumptions, however, have a different role. Not only do they imply some guarantees, but in addition any design satisfying the considered contract *must only be used in an environment meeting* the strong assumptions. Said differently, an environment failing to meet the strong assumption cannot host the considered component. Thus, strong assumptions are related somehow to the issue of compatibility.

The pedagogical value of strong versus weak assumptions was extensively discussed in our companion paper [16], in the two sections on Platform Based Design and on AUTOSAR. One important objective is to be able to combine, under conjunction, “weak contracts”, which are implications $\{\text{assumption}\} \Rightarrow \{\text{guarantee}\}$, and “strong contracts”, in which the environment must satisfy the assumption. Strong assumptions must be met, whereas the weak contract vanishes if its assumptions are not met by the environment.

An Assume/Guarantee contract is thus a tuple

$$\mathcal{C} = (A, B; G, H)$$

of properties, where

- A and B denote the *strong* and *weak assumptions* and
- G and H denote the *strong* and *weak guarantees*,

respectively. In the following we examine the details of this extended Assume/Guarantee contract framework, which subsumes the previous variants.

B. Handling dissimilar alphabets

Since the alphabet Σ of actions, signals, and/or variables varies with the design, assumption, or guarantee, it is important to define how sets of behaviors are extended to a larger alphabet $\Sigma' \supset \Sigma$. This is performed by using the classical mechanism of inverse projections. Say that a behavior s' over Σ' *extends* a behavior s over Σ if erasing, in s' , occurrences labeled by elements of $\Sigma' - \Sigma$ yields s . Then, for M an assertion with alphabet Σ and $\Sigma' \supset \Sigma$, we define $M^{\uparrow\Sigma'}$ as being the set of behaviors over Σ' that are extensions, to Σ' , of behaviors of M . Extending alphabets allows us to compare and combine, using set theoretic operators, two assertions M_1 and M_2 with respective alphabets Σ_1 and Σ_2 , by considering $\Sigma \triangleq \Sigma_1 \cup \Sigma_2$ and then $M_1^{\uparrow\Sigma}$ and $M_2^{\uparrow\Sigma}$, which are now subsets of the same set of all possible behaviors over alphabet Σ . The same mechanism of alphabet equalization will be used for assumptions and guarantees. Thus, in the sequel, expressions such as $A \wedge A'$ or $M \times N$ implicitly assume that alphabet equalization has been performed prior to applying the referred logical or set theoretic operation.

C. The framework and how it instantiates the meta-theory

Say that

$$E \text{ is a compatible environment for } \mathcal{C} \text{ if } E \subseteq A. \quad (10)$$

The intent is that a component satisfying contract \mathcal{C} can only be used in the context of a compatible environment. The *implementation* relation $M \models \mathcal{C}$ is defined as follows:

$$\forall E, E \text{ compatible} \quad \text{then} \quad E \times M \subseteq G \quad (11)$$

$$\text{if in addition } E \subseteq B \quad \text{then} \quad E \times M \subseteq G \wedge H \quad (12)$$

The difference between strong and weak assumptions is that the former are mandatory for any compatible environment.

For \mathcal{C} a contract, we are not interested in its particular form, but only in the set of its correct implementations and compatible environments. Thus, two contracts \mathcal{C} and \mathcal{C}' are considered *equivalent*, written $\mathcal{C} \sim \mathcal{C}'$, if they possess identical sets of correct implementations:

$$\mathcal{C} \sim \mathcal{C}' \quad \text{iff} \quad \begin{cases} A' = A \\ G' \vee \neg A' = G \vee \neg A \\ H' \vee \neg B' = H \vee \neg B \end{cases}$$

Any contract can be put in an equivalent *canonical form*:

$$\mathcal{C} = (A, T; G, H), \text{ where } G \supseteq \neg A$$

Canonical forms are not unique. It is unnecessary to keep the trivial assumption T (“true”), so we remove it and work instead with triples of the form $\mathcal{C} = (A; G, H)$ in canonical form, i.e., such that $G \supseteq \neg A$ or, equivalently, $\neg G \subseteq A$.

Following (3) in Table II summarizing the meta-theory, contract *refinement* is by inclusion of corresponding sets of correct implementations. To ensure substitutability with respect to the environments, we also require that the set of compatible environments is extended via contract refinement. Thus, for \mathcal{C} and \mathcal{C}' in canonical form,

$$\mathcal{C}' \leq \mathcal{C} \quad \text{iff} \quad \begin{cases} A' \supseteq A \\ G' \subseteq G \\ H' \subseteq H \end{cases} \quad (13)$$

Conjunction $\mathcal{C}' \wedge \mathcal{C}$ follows as the greatest lower bound of refinement. Conjunction preserves canonical forms.

We want that *parallel composition* implements definition (5) of the meta-theory of Table II, or at least that it approximates it in a sound way, thus supporting independent development. We thus define the parallel composition as follows, for two contracts \mathcal{C}_1 and \mathcal{C}_2 in canonical form:

$$\mathcal{C}_1 \otimes \mathcal{C}_2 = (A; G, H) : \begin{cases} G = G_1 \wedge G_2 \\ H = H_1 \wedge H_2 \\ A = (A_1 \wedge A_2) \vee \neg(G_1 \wedge G_2) \end{cases} \quad (14)$$

Lemma 1: Formula (14) defines a contract in canonical form. Composition \otimes is associative, commutative, and idempotent. In addition, the following holds, for any two contracts \mathcal{C}_1 and \mathcal{C}_2 and two designs M_1 and M_2 such that $M_i \models \mathcal{C}_i$ for $i = 1, 2$:

$$E \subseteq A \Rightarrow \begin{cases} M_2 \times E \subseteq A_1 \\ M_1 \times E \subseteq A_2 \end{cases} \quad (15)$$

$$M_1 \times M_2 \models \mathcal{C}_1 \otimes \mathcal{C}_2 \quad (16)$$

Proof: The first statement follows from $G \vee A = (G_1 \wedge G_2) \vee (A_1 \wedge A_2) \vee \neg(G_1 \wedge G_2) = T$. Next, since $M_2 \subseteq G_2$ and using the fact that contracts \mathcal{C}_1 and \mathcal{C}_2 are in canonical form:

$$\begin{aligned} E \times M_2 &\subseteq E \times G_2 \\ &\subseteq A \times G_2 \\ &= ((A_1 \wedge A_2) \vee \neg(G_1 \wedge G_2)) \wedge G_2 \\ &= (A_1 \wedge A_2 \wedge G_2) \vee ((\neg G_1 \vee \neg G_2) \wedge G_2) \\ &= (A_1 \wedge A_2 \wedge G_2) \vee (\neg G_1 \wedge G_2) \\ &\subseteq (A_1 \wedge A_2 \wedge G_2) \vee (A_1 \wedge G_2) \\ &\subseteq A_1 \end{aligned}$$

which proves (15). To prove (16), pick E compatible for \mathcal{C} , i.e., $E \subseteq A$. Since contracts are in canonical form, $E \times M_1 \times M_2 \subseteq A \wedge G_1 \wedge G_2 \subseteq G_1 \wedge G_2$, which proves (11). Property (12) is proved similarly. Hence (16) follows. \diamond

Compatibility is addressed in this framework, through the consideration of *strong assumptions*. For $\mathcal{C} = (A, B; G, H)$ a contract — not necessarily in canonical form — and M a design implementing it, we have defined an environment E to be *compatible* with M if and only if E meets the strong assumption A . Note that this allows for a contract to expose which environment is compatible with it, see the discussion section III-C. Lemma 1 expresses that this notion of compatibility is propagated by the parallel composition \otimes for contracts: if \mathcal{C}_1 and \mathcal{C}_2 are two contracts, then the strong assumption of $\mathcal{C}_1 \otimes \mathcal{C}_2$ defines compatible environments for

this composition. And (15) expresses that, in this composition, $E \times M_1$ is a compatible environment for M_2 and symmetrically. Now, for A defined in (14), $A = F$ expresses that the two contracts are *incompatible*. Since operation \otimes is associative and commutative, this way of addressing compatibility extends to a set of components for composition.

On the other hand, no notion of quotient following (8) in Table II is offered by this framework.

D. Answering the questions raised in Section II on the parking garage example

At this point we are ready to answer the questions raised in Section II on the parking garage example.

Consider first Property 3 regarding refinement and the first question raised. Specification 1 defines the overall system, including the user. This system S is therefore closed and its specification is a contract $\mathcal{C} = (T, T; G, G)$, where strong and weak assumptions are trivial (T) and strong and weak guarantees are identical and equal to the conjunction G of all listed requirements. Now, consider Contracts 1 to 3 and interpret the listed assumptions as being strong ones — the concepts of weak and strong assumptions were not used in the example, so this choice is arbitrary and for illustration purposes. We can thus ignore the weak assumptions and guarantees and describe the above (strong) contracts as pairs (A_1, G_1) , (A_2, G_2) , and (A_3, G_3) , respectively. We then observe that $G = G_1 \wedge G_2 \wedge G_3$, since every requirement listed in Specification 1 is also listed as a guarantee in one of the three contracts. Regarding the assumptions, we have $(A_1 \wedge A_2 \wedge A_3) \vee \neg(G_1 \wedge G_2 \wedge G_3) = (A_1 \wedge A_2 \wedge A_3) \vee \neg G = (A_1 \wedge A_2 \wedge A_3) \vee \neg(A_1 \wedge A_2 \wedge A_3) = T$, since every guarantee of the overall system becomes an assumption for one of the contracts. Therefore, refinement holds. This reasoning applies to any well formed allocation of requirements to subsystems, taking responsibilities carefully into account.

Consider next the second question stated in Property 3. Here, refinement does not hold since assumptions were reinforced when moving from Contract 2 to Contract 4.

Finally, focus on the discussion following Question 1 regarding conjunction of viewpoints. The explanations in this paragraph indicate that assumptions were meant to be *weak* in conjunction $\mathcal{C}_{\text{park_all_viewpoints}} = \mathcal{C}_{\text{park}} \wedge \mathcal{C}_{\text{park_safety}}$, since it is intended that each of the two contracts survives if its own assumptions are met, even if assumptions of the other contract are violated. We could, instead, consider that functional contract $\mathcal{C}_{\text{park}}$ is strong whereas safety contract $\mathcal{C}_{\text{park_safety}}$ is weak. Requiring that contract $\mathcal{C}_{\text{park_safety}}$ is strong would set hard constraints on the execution platform of the parking.

E. Practical implementation

Here we sketch the way Assume/Guarantee contracts can be implemented in practice by using (almost) only tests. Suppose we know how to map any property P to an associated test T (see Definition 1) in the form of a deterministic machine taking as inputs the signals and variables involved in P and

producing a boolean indicating satisfaction or violation of the requirement. We have mentioned PSL as a practical way to express properties. In the case of SPEEDS contracts, requirements were expressed in CSL, a formal textual language of patterns that can be translated into a particular class of automata. But other formalisms can be used as well for tests, see Section IV.

Now, we explain how we can check whether a design M satisfies a guarantee G when put in a context satisfying assumption A . Our discussion will cover both cases of assumption A being strong or weak. Consider the case in which assumption A and guarantee G consist of the conjunction of properties $(R_{A,i})_{i=1..n}$ and $(R_{G,j})_{j=1..m}$, respectively. Consider the associated tests $T_{A,i}$ for $i = 1 \dots n$ and $T_{G,j}$ for $j = 1 \dots m$, with corresponding outputs $B_{A,i}$ for $i = 1 \dots n$ and $B_{G,j}$ for $j = 1 \dots m$, respectively. Then, checking whether M satisfies a guarantee G when put in a context satisfying assumption A is performed as follows:

- Using tests $T_{A,i}$, $i = 1 \dots n$, filter out the illegal histories of signals and variables of the design M by keeping only the histories that are correct with respect to the assumptions, i.e., such that the conjunction $B_A \triangleq \bigwedge_{i=1}^n B_{A,i}$ never switches to F. If A is strong, then existence of runs of the environment violating B_A leads to rejecting this environment.
- Submit the above correct histories to all tests $T_{G,j}$ and check for their satisfaction, i.e., check that $B_G \triangleq \bigwedge_{j=1}^m B_{G,j}$ never switches to F.

If assumption A is weak, we can simply submit the histories of signals and variables of the design M by 1/ submitting them to all tests $T_{A,i}$ and $T_{G,j}$, 2/ evaluating the boolean variable

$$B_C \triangleq \underbrace{\left(\bigwedge_{j=1}^m B_{G,j} \right)}_{B_G} \vee \neg \underbrace{\left(\bigwedge_{i=1}^n B_{A,i} \right)}_{B_A} \quad (17)$$

Boolean B_C can be evaluated on-line, with, however, the convention that it sticks to F as soon as it reaches this status.

Thus, a contract $\mathcal{C} = (A; G, H)$ is represented by the triple of booleans $(B_A; B_G, B_H)$, where B_A and B_G are defined in the underbrackets of (17) and B_H is defined similarly. By abuse of notation, we simply write

$$\mathcal{C} = (B_A; B_G, B_H) \quad (18)$$

So far the above simply parallels what we did for observers, see Section IV. The novelty with respect to observers comes next. Using representation (18), we can re-express the basic relations and operations on contracts:

$$\begin{aligned} \mathcal{C}_1 \wedge \mathcal{C}_2 &= (B_{A_1} \vee B_{A_2}; \\ &\quad B_{G_1} \wedge B_{G_2}, B_{H_1} \wedge B_{H_2}) \\ \mathcal{C}_1 \otimes \mathcal{C}_2 &= (B_A; B_G, B_H), \text{ where} \\ B_A &= [B_{A_1} \wedge B_{A_2}] \vee \neg[B_{G_1} \wedge B_{G_2}] \\ B_G &= [B_{G_1} \wedge B_{G_2}] \\ B_H &= [B_{H_1} \wedge B_{H_2}] \end{aligned} \quad (19)$$

$$\mathcal{C}' \text{ refines } \mathcal{C} \text{ if } B_{A'} \Leftarrow B_A \text{ and } B_{G'} \Rightarrow B_G$$

Formulas (19) provide a full fledged calculus of contracts supporting all desired features listed in Section III. The practical implementation consists in maintaining a data base of tests attached to each single requirement (very much like the tests that are attached to requirements in the traditional approach). Then, the formulas (19) are used to answer the various questions related to contract based design, see Section II-E. In particular, to test for correctness, $M \models \mathcal{C}$, we represent \mathcal{C} as in (18) and then, using formulas (19), expand B_A, B_G , and B_H using boolean expressions involving the boolean output by the primitive tests T_{A_i}, T_{G_j} , and T_{H_k} attached to assumptions and guarantees. It then suffices to feed each of these tests with inputs consisting of variables and events input to or output by design M .

F. The problem with handling dissimilar alphabets

The handling of dissimilar alphabets as explained in Section V-B raises a serious problem. The difficulty originates from the definition (13) for refinement, from which the definition of the conjunction follows. Consider a conjunction $\mathcal{C} = \mathcal{C}_1 \wedge \mathcal{C}_2$ in which $\mathcal{C}_1 = (\mathbb{T}; B_{G_1}, B_{H_1})$ has a trivial strong assumption. Then, \mathcal{C} inherits this trivial assumption: $\mathcal{C} = (\mathbb{T}; B_G, B_H)$. Now, suppose in addition that the above two contracts involve disjoint sets of variables or events, from a same component — for example, \mathcal{C}_1 is functional whereas \mathcal{C}_2 relates to response time (e.g., “if inputs to the component are submitted below a given rate, then response time is guaranteed to stay within some bound”). Then, it is certainly too strong considering that the conjunction of these two contracts has a guarantee that must hold under any environment: guaranteeing the response time for any input rate is an unachievable guarantee. The same artifact arises even if the assumption is non-trivial in \mathcal{C}_1 . Suppose again that the two contracts involve disjoint sets of variables or events and the environment meets the assumption of \mathcal{C}_1 , i.e., $B_{A_1} = \mathbb{T}$ for that environment. Then, again, for that environment, $B_A = \mathbb{T}$ follows, which means that the guarantee of the second contract should be met regardless of the satisfaction of its corresponding assumption, which might be too demanding in practice. As a consequence, Assume/Guarantee contracts as developed in this section will not satisfy the wanted conditions for the conjunction of the functional and safety viewpoints for the gates, see the discussion following the safety Requirements 5.

G. Extensions

So far Assume/Guarantee contracts were presented in a general setting. Assertions are simply sets of behaviours, and thus as such can carry different aspects such as the functions being performed, but also timing characteristics, energy characteristics, and so on. Observe that the practical implementation suggested in Section V-E relies on testing, and is thus applicable to any type of system — obviously with the inherent limitations of test based approaches. In this section the extensions we review should be rather called “focalisations” in that the reasoning that is proposed is tailored to specific aspects.

1) *Time*: A notion of contract for real-time interfaces is proposed in [24]. 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.

2) *Stochastic*: The stochastic nature of embedded systems has to be accounted for during design. For instance, certification agencies require that designers of airborne embedded systems provide sufficient evidence that precise system-level reliability requirements are satisfied. In many other instances, reliability analysis is an important milestone, despite it is not part of a certification process. Until now, this is done late in the design process. The main reason is that state of the art reliability analysis methods lack modularity and, thus, do not apply to partial designs.

In a recent PhD work, an assume/guarantee contract theory has been extended to a stochastic setting [55], [56], [57]. 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 $\alpha_1 + \alpha_2 - 1$.

H. Discussion

By explicitly manipulating assumptions and guarantees, Assume/Guarantee (A/G) contracts instantiate the intuition that the designer must specify explicit assumptions for the context of use of her design. Also, the theory is extremely simple when the properties considered (assumptions or guarantees) are sets of traces or languages. This includes the class of safety properties [88], [41], [102], [8], [42]. Designs are then seen as the set of traces they can generate when executing them. Strong and weak assumptions offer added value. So far

such theories have been favored when advocating the use of contracts by the industry.

There are, however, technical reasons for not being fully satisfied with A/G contracts. While the distinction between weak and strong assumptions provides a way of addressing compatibility in some indirect way, its practical use may be subtle. Finally, conjunction is problematic when applied to contracts with different alphabets of actions and variables, a situation typically encountered when designing large systems.

VI. INTERFACES AS CONTRACTS

In contrast to the framework of Assume/Guarantees contracts, *Interface Theories* [52], [39], [92], [61], [83] do not keep an explicit distinction between assumptions and guarantees.

A. Interfaces and Modalities

Modalities: 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, e.g., the obligation to perform some action in a given situation. Modalities were proposed by Kim Larsen [95], [6], [30] as a simple and elegant framework to express both allowed and mandatory properties. As an illustration, focus on requirement $R_{1.3}$ from the parking garage example of Section II, which involves both keywords “may” and “must”. Figure 10 shows how to express this requirement using an

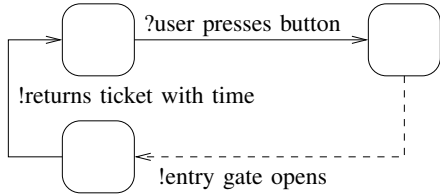


Figure 10. Requirement $R_{2.3}$ using modalities.

automaton with modalities. Dashed transitions indicate moves that *may* be taken, whereas solid transitions indicate moves that *must* be taken. Observe that a solid transition is used for “?user presses button”. This does not mean that a user must press the button, but it means that the gates *must accept* that a user presses the button: the gates cannot refuse users to press the entry button. Thus, the interpretation of solid/dashed differs between events labeled “!”, which are under the control of the gates, and events labeled by a “?”, which are under the control of the environment of the gates. This seems like a minor improvement with respect to the other formalisms. It turns out to be a fundamental one, however.

Modal Specifications: 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 [95], [90] in order to allow for loose specifications of systems. Since then, they have been considered for automata [93] and formal languages [113] and applied to a wide range of application domains (see [6] for

a complete survey). Informally, a *must* transition is available in every component that implements the modal specification, while a *may* transition needs not be. A modal specification thus represents a set of implementations.

Denote by **may** the set of transitions having *may* as a modality and by **must** the set of transitions having *must* as a modality. The following *consistency* condition is required:

$$\mathbf{must} \subseteq \mathbf{may} \quad (20)$$

It naturally imposes that every required transition is also allowed. However, when combining specifications, we shall see that this consistency condition may get violated as discrepancies between the modal information carried out by the specifications may appear. For this purpose, mixed specifications (also called pseudo-modal specifications in [114]) have been introduced. In Modal Specifications no particular attention is paid to the different roles of component versus environment or input versus output. Hence, not surprisingly, issues of compatibility cannot be addressed by this framework.

Interface Automata: In contrast, Interface Automata, proposed by de Alfaro and Henzinger [52], [50], [4], [37], focus primarily on compatibility. Interface Automata are automata whose transitions are labeled with *input* and *output* actions. Input actions (marked by a “?”) are under the responsibility of the environment whereas output actions (marked by a “!”) are under the responsibility of the component itself. The framework of Interface Automata does not handle assumptions and guarantees explicitly; however it captures the different responsibilities by carefully distinguishing the *component* from its *environment*.

In contrast to Nancy Lynch’s Input-Output automata [101], Interface Automata are not necessarily input-enabled, i.e., they are not required to accept any input action in all states. In particular, this allows one to express assumptions on the environment as the environment is supposed not to submit an action that is not enabled in the current state. Suppose input action $a?$ is submitted by the environment while the component is in state s and a is not enabled at s . This raises an exception. Two ways of handling this exception can be considered:

- *Strong approach:* exceptions are refused. Then the above situation yields a run-time error, which is a violation of compatibility. Accordingly, if the strong approach is followed, such an environment cannot interact with the considered component.
- *Weak approach:* exceptions are accepted but result in exception handling. In the resulting degraded mode, guarantees offered by the component are no longer met. Accordingly, if the weak approach is followed, such an environment can interact with the component but causes the latter to switch to some degraded mode in which the guarantees offered by the component are relaxed.

Interface Automata follow the strong approach and address compatibility in depth. Indeed, their whole purpose was to advocate the importance of compatibility and our discussion

of Section III-A builds on their original arguments. Also, these authors were the first to pinpoint the need for revisiting abstraction as discussed in Section III-A.

Interface Automata are operational models, they do not encompass any notion of implementation, and thus neither satisfiability nor consistency, because one cannot distinguish between interfaces and component implementations. Properties of interfaces are described in game-based logics, e.g., ATL [5], with a theoretical high-cost complexity. The semantics of an Interface Automaton is 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).

The main advantage of the game-based approach appears in the definition of composition and *compatibility* between components. Referring to our general discussion of Section III-A, compatibility in Interface Automata addresses deadlock, an important kind of run-time error. Following [50], two interface automata are composable if they possess disjoint sets of output actions, compose by synchronizing on shared actions, and interleave asynchronously all other actions. Since interface automata are not necessarily input-enabled, in the product $\mathcal{P}_1 \times \mathcal{P}_2$ of two interface automata \mathcal{P}_1 and \mathcal{P}_2 , there may be pairs of states (s_1, s_2) which are *illegal* in the sense that one of the automata may produce an output action that is also in the input alphabet of the other automaton, but is not accepted at this state. If (s_1, s_2) is reachable in the product $\mathcal{P}_1 \times \mathcal{P}_2$ seen as automata, then deadlock can occur in this product. In previous existing models for interface theories based on an input/output setting, such interfaces are pessimistically declared *incompatible*. Now, by exploiting the game-based semantics an optimistic approach for compatibility is possible:

“Two interfaces can be composed and are compatible if there is at least one environment where they can work together (i.e., where they can avoid the illegal states).” [52]

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 [52], [39]. Deciding whether there exists an environment where the two interfaces can work together is equivalent to checking whether the environment in the product of the interfaces has a strategy to always avoid illegal states. Applying on $\mathcal{P}_1 \times \mathcal{P}_2$ the pruning of illegal states that results yields what the authors call the *parallel composition* $\mathcal{P}_1 \parallel \mathcal{P}_2$ of the two interfaces. Computing this is a problem of *controller synthesis* in the Ramadge-Wonham sense [34]. Other aspects of compatibility are not addressed by Interface Automata. In [51] the range of compatibility is extended by proposing *social interfaces*, where interface models can communicate via both actions and shared variables, and where communication and synchronization covers the full spectrum, from one-to-one, to one-to-many, many-to-one, and many-to-many; see also [99].

From the point of view of the component, *refinement*

amounts to offering stronger guarantees while relaxing the obligations of the environment. Formally, refining an Interface Automaton \mathcal{P} into \mathcal{P}' is by *alternate simulation*, defined as follows: a state s' of \mathcal{P}' *simulates* a state s of \mathcal{P} if 1/ whatever output \mathcal{P}' can emit in state s' can also be output by \mathcal{P} in state s , and 2/ whatever input \mathcal{P} accepts in state s is also accepted by \mathcal{P}' in state s' , and 3/ performing such identical moves on \mathcal{P} and \mathcal{P}' leads to two states s_1 and s'_1 such that s'_1 simulates s_1 . Then, \mathcal{P}' *refines* \mathcal{P} if the initial state of \mathcal{P}' simulates the initial state of \mathcal{P} .

In [23], incremental design of deterministic Interface Automata is studied. Let $\widehat{\mathcal{P}}$ be the interface \mathcal{P} with input and output actions interchanged. Given two Interface Automata \mathcal{P} and \mathcal{P}_1 , the greatest interface compatible with \mathcal{P}_1 such that their composition refines \mathcal{P} is given by

$$\widehat{\mathcal{P}_1 \parallel \widehat{\mathcal{P}}}$$

Finally, *software interfaces* were proposed in [38], as a push-down 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,¹⁸ an operating system for sensor networks.

Despite the availability of partial results [61], the framework of Interface Automata lacks support for the conjunction of interfaces in a general setting, and does not offer the flexibility of modalities. It is thus a partial instantiation of the meta-theory of contracts presented in Section III, dealing only with compatibility in which a runtime error corresponds to the reachability of an illegal state.

Combining Modal Specifications and Interface Automata yields Modal Interfaces: Since assumptions concern the environment and guarantees concern the component, we need to be able to distinguish between the two. However no provision is offered for such a distinction in Modal Specifications. *Modal Interfaces* combine the modalities of Modal Specifications with the distinction between *input* (action controlled by the environment) and *output* (action controlled by the component) and handle them in the same way as in Interface Automata [92], [83]. The next sections describe the framework of Modal Interfaces and how the extensions are able to handle the requirements of contract theory.

B. The framework and how it instantiates the meta-theory

Throughout this section, we first suppose that all Modal Interfaces are defined over a fixed alphabet Σ of events. This restriction will be removed in Section VI-C where the handling of dissimilar alphabets is addressed using the technique of alphabet equalization.

Modal Interfaces are tuples $\mathcal{C} = (\Sigma, S, s^0, may, must)$, where Σ is a finite alphabet of valued *events* whose domain may be infinite; S is a possibly infinite set of *states* and $s^0 \in S$ is the *initial state*; $may, must \subseteq S \times \Sigma \rightarrow S$ are the *may* and

¹⁸<http://www.tinyos.net/>

\wedge	$s_2 \xrightarrow{\sigma} s'_2$	$s_2 \xrightarrow{-\sigma} s'_2$	$s_2 \xrightarrow{\sigma} \perp$
$s_1 \xrightarrow{\sigma} s'_1$	$(s_1, s_2) \xrightarrow{\sigma} (s'_1, s'_2)$	$(s_1, s_2) \xrightarrow{\sigma} (s'_1, s'_2)$	$(s_1, s_2) \xrightarrow{\sigma} \perp$
$s_1 \xrightarrow{-\sigma} s'_1$	$(s_1, s_2) \xrightarrow{\sigma} (s'_1, s'_2)$	$(s_1, s_2) \xrightarrow{-\sigma} (s'_1, s'_2)$	$(s_1, s_2) \xrightarrow{\sigma} \perp$
$s_1 \xrightarrow{\sigma} \perp$	$(s_1, s_2) \xrightarrow{\sigma} \perp$	$(s_1, s_2) \xrightarrow{\sigma} \perp$	$(s_1, s_2) \xrightarrow{\sigma} \perp$

Table III
TRANSITIONS RELATIONS OF $\mathcal{C}_1 \& \mathcal{C}_2$

\otimes	$s_2 \xrightarrow{\sigma} s'_2$	$s_2 \xrightarrow{-\sigma} s'_2$	$s_2 \xrightarrow{\sigma} \perp$
$s_1 \xrightarrow{\sigma} s'_1$	$(s_1, s_2) \xrightarrow{\sigma} (s'_1, s'_2)$	$(s_1, s_2) \xrightarrow{-\sigma} (s'_1, s'_2)$	$(s_1, s_2) \xrightarrow{\sigma} \perp$
$s_1 \xrightarrow{-\sigma} s'_1$	$(s_1, s_2) \xrightarrow{-\sigma} (s'_1, s'_2)$	$(s_1, s_2) \xrightarrow{-\sigma} (s'_1, s'_2)$	$(s_1, s_2) \xrightarrow{\sigma} \perp$
$s_1 \xrightarrow{\sigma} \perp$	$(s_1, s_2) \xrightarrow{\sigma} \perp$	$(s_1, s_2) \xrightarrow{\sigma} \perp$	$(s_1, s_2) \xrightarrow{\sigma} \perp$

Table IV
TRANSITIONS RELATIONS OF $\mathcal{C}_1 \otimes \mathcal{C}_2$

\parallel	$s_2 \xrightarrow{\sigma} s'_2$	$s_2 \xrightarrow{-\sigma} s'_2$	$s_2 \xrightarrow{\sigma} \perp$
$s_1 \xrightarrow{\sigma} s'_1$	$(s_1, s_2) \xrightarrow{\sigma} (s'_1, s'_2)$	$(s_1, s_2) \xrightarrow{\sigma} \perp$	$(s_1, s_2) \xrightarrow{\sigma} \perp$
$s_1 \xrightarrow{-\sigma} s'_1$	$(s_1, s_2) \xrightarrow{-\sigma} (s'_1, s'_2)$	$(s_1, s_2) \xrightarrow{-\sigma} (s'_1, s'_2)$	$(s_1, s_2) \xrightarrow{-\sigma} \top$
$s_1 \xrightarrow{\sigma} \perp$	$(s_1, s_2) \xrightarrow{\sigma} \perp$	$(s_1, s_2) \xrightarrow{\sigma} \perp$	$(s_1, s_2) \xrightarrow{-\sigma} \top$

Table V
TRANSITIONS RELATIONS OF $\mathcal{C}_1 \parallel \mathcal{C}_2$

must relations. The consistency condition (20) now becomes: for any $s \in S$ and $\sigma \in \Sigma$,

$$\text{must}(q, \sigma) \subseteq \text{may}(q, \sigma) \quad (21)$$

We shall write $s \xrightarrow{\sigma} s'$, $s \xrightarrow{-\sigma} s'$ and $s \xrightarrow{\sigma} \perp$ to mean $s' \in \text{must}(s, \sigma)$, $s' \in \text{may}(s, \sigma) \setminus \text{must}(s, \sigma)$ and $\text{may}(s, \sigma) = \emptyset$, respectively. Whenever required, events are labeled by “?” (input) or “!” (output) to indicate that the event is uncontrolled or controlled by the considered component. We denote by $\Sigma_?, \Sigma_! \subseteq \Sigma$ the sets of inputs and outputs of \mathcal{C} . They form a partition of Σ and are called the *signature* of \mathcal{C} .

Models of Modal Interfaces, called *Designs*, are tuples $M = (\Sigma, Q, i^0, \rightarrow)$, where Σ is a finite alphabet of valued events; Q is a set of states and $i^0 \in Q$ is the initial state; $\rightarrow \subseteq Q \times \Sigma \times Q$ is the transition relation. Write $q \xrightarrow{\sigma} q'$ and $q \xrightarrow{-\sigma} q'$ if $(q, \sigma, q') \in \rightarrow$ and if there is no q' such that $(q, \sigma, q') \in \rightarrow$, respectively. Designs are equipped with a parallel composition operator $M_1 \times M_2$ defined as the tuple $M = (\Sigma, Q_1 \times Q_2, (i_1^0, i_2^0), \rightarrow)$ where $((q_1, q_2), \sigma, (q'_1, q'_2)) \in \rightarrow$ if and only if $(q_1, \sigma, q'_1) \in \rightarrow_1$ and $(q_2, \sigma, q'_2) \in \rightarrow_2$.

We now discuss how Modal Interfaces can handle the key operations of the meta-theory of contracts presented in Section III.

Implementation: A Design M is correct with respect to a Modal Interface \mathcal{C} if any action labeled *must* in \mathcal{C} is also offered by M , and only actions labeled *may* or *must* in \mathcal{C} are offered by M . If we consider infinite state systems, checking whether $M \models \mathcal{C}$ can be done by taking every pair of states (q, s) , where q is from M and s from \mathcal{C} , reachable after some common behavior, and by performing a *double test* that corresponds to the informal conditions outlined above. A violation of the implementation relation $M \models \mathcal{C}$ is detected if there exists an event σ that corresponds to a move from q of

design M which is not allowed by \mathcal{C} in s , or such that there is no move from q in M , but a move is required by \mathcal{C} in s . More formally, a violation of the implementation relation $M \models \mathcal{C}$ can be detected by checking the following conditions on every pair (q, s) of states reachable after some common behavior:

$$\begin{aligned} [q \xrightarrow{\sigma} q'] \wedge [s \xrightarrow{\sigma}] & \text{ yields } \neg[M \models \mathcal{C}] \\ [q \xrightarrow{-\sigma}] \wedge [s \xrightarrow{\sigma} s'] & \text{ yields } \neg[M \models \mathcal{C}] \end{aligned} \quad (22)$$

Refinement: Modal refinement consists in deleting some *may* transition and turning other *may* transitions into *must*. Refinement between modal interfaces can also be checked via *double testing*. Let \mathcal{C}_1 and \mathcal{C}_2 be two modal interfaces. A violation of the refinement relation $\mathcal{C}_1 \leq \mathcal{C}_2$ can be detected by checking the following conditions on every pair (s_1, s_2) of states reachable after some common behavior:

$$\begin{aligned} [s_1 \xrightarrow{-\sigma} s'_1 \vee s_1 \xrightarrow{\sigma} s'_1] \wedge [s_2 \xrightarrow{\sigma}] & \text{ yields } \neg[\mathcal{C}_1 \leq \mathcal{C}_2] \\ [s_1 \xrightarrow{-\sigma} s'_1 \vee s_1 \xrightarrow{\sigma}] \wedge [s_2 \xrightarrow{\sigma} s'_2] & \text{ yields } \neg[\mathcal{C}_1 \leq \mathcal{C}_2] \end{aligned} \quad (23)$$

Recall that the alphabet Σ is fixed. Define the particular Modal Interfaces $\mathcal{C}_\top = (\Sigma, \{s^0\}, s^0, \text{may}, \text{must})$ with $s^0 \in \text{may}(s^0, \sigma)$ for all $\sigma \in \Sigma$ and $\text{must}(s^0, \sigma) = \emptyset$. This Modal Interface admits any design as model. Conversely, we introduce the Modal Interfaces \mathcal{C}_\perp whose set of states S is empty and which admits no design. Observe that for any Modal Interfaces \mathcal{C} , we have $\mathcal{C}_\perp \leq \mathcal{C} \leq \mathcal{C}_\top$.

Conjunction: Modal Specifications offer built-in conjunction of specifications [94], [115]. The conjunction of Modal Interfaces \mathcal{C}_1 and \mathcal{C}_2 over the same signature is defined as their greatest lower bound (GLB) $\mathcal{C}_1 \wedge \mathcal{C}_2$ with respect to the refinement order \leq . Given two Modal Interfaces $\mathcal{C}_1 = (\Sigma, S_1, s_1^0, \text{may}_1, \text{must}_1)$ and $\mathcal{C}_2 = (\Sigma, S_2, s_2^0, \text{may}_2, \text{must}_2)$, we first construct a pre-conjunction denoted $\mathcal{C}_1 \& \mathcal{C}_2 = (\Sigma, (S_1 \times S_2) \cup \{\perp\}, (s_1^0, s_2^0), \text{may}, \text{must})$ with the transitions

given by Table III. For \perp , we let $must(\perp, \sigma) = \perp$ and $may(\perp, \sigma) = \emptyset$ for all $\sigma \in \Sigma$, hence \perp violates the consistency condition (21); this particular state serves as an artifice to model discrepancies between the modal informations carried out by the interfaces. Observe, in particular, that when $s'_1 \in must_1(s_1, \sigma)$ and $may_2(s_2, \sigma) = \emptyset$ then $\perp \in must((s_1, s_2), \sigma)$. Then the GLB $\mathcal{C}_1 \wedge \mathcal{C}_2$ is obtained from $\mathcal{C}_1 \& \mathcal{C}_2$ by backward killing any may-transition from any state (s'_1, s'_2) to (s_1, s_2) . This may cause new inconsistencies as the source state (s'_1, s'_2) may become inconsistent if there is a must-transition to (s_1, s_2) . Thus the process must be iterated until stabilization (see [115] for details). The conjunction is a sound approximation (but not complete in general, see below) of the operation (4) in Table II of the meta-theory.

The “merge” of non-deterministic Modal Specifications regarded as partial models has been considered in [124]. This operation consists in looking for common refinements of initial specifications and is thus similar to the conjunction operation presented here. In [124], [67], 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* [60].

Parallel composition: Modal interfaces are equipped with a parallel composition $\mathcal{C}_1 \otimes \mathcal{C}_2$ which consists in synchronizing transitions having the same labels and assigning a modality *may*; a modality *must* is also assigned provided that the synchronized transitions have both the modality *must*. Moreover, inputs and outputs are synchronized: if one interface produces an output $!\sigma$ and the other one accepts it as an input $?\sigma$ then the resulting transition in the parallel composition is labeled $!\sigma$.

More formally, given $\mathcal{C}_1 = (\Sigma, S_1, s_1^0, may_1, must_1)$ and $\mathcal{C}_2 = (\Sigma, S_2, s_2^0, may_2, must_2)$, we construct the parallel product $\mathcal{C}_1 \otimes \mathcal{C}_2 = (\Sigma, S_1 \times S_2, (s_1^0, s_2^0), may, must)$ with the transitions given by Table IV. Observe that the parallel composition does not generate states that violate condition (20).

Quotient: The quotient of Modal Specifications has been largely studied in [89], [114]. It provides a dual to the parallel composition and plays a role for incremental design of systems. Suppose one wants to realize the contract \mathcal{C}_1 by reusing a preexisting component implementing \mathcal{C}_2 . Then $\mathcal{C}_1/\mathcal{C}_2$ is the largest Modal Interface such that $(\mathcal{C}_1/\mathcal{C}_2) \otimes \mathcal{C}_2$ refines \mathcal{C}_1 . Equivalently, it is the sufficient and necessary condition that a design M_1 must satisfy in order for $M_1 \times M_2$ to correctly implement \mathcal{C}_1 whenever M_2 correctly implements \mathcal{C}_2 .

Given two modal interfaces \mathcal{C}_1 and \mathcal{C}_2 , we first construct a pre-quotient $\mathcal{C}_1 // \mathcal{C}_2 = (\Sigma, (S_1 \times S_2) \cup \{\perp, \top\}, (s_1^0, s_2^0), may, must)$ with the transitions given by Table V. For \top , we let $must(\top, \sigma) = \emptyset$ and $may(\top, \sigma) = \top$ for all $\sigma \in \Sigma$. For \perp , we let $must(\perp, \sigma) = \perp$ and $may(\perp, \sigma) = \emptyset$ for all $\sigma \in \Sigma$, hence \perp violates the consistency condition (21). Observe, in particular, that when $s'_1 \in must_1(s_1, \sigma)$ then it is expected that $s'_2 \in must_2(s_2, \sigma)$ otherwise \perp is reachable. Thus, the quotient $\mathcal{C}_1/\mathcal{C}_2$ is built from $\mathcal{C}_1 // \mathcal{C}_2$ by applying the same pruning operation as was done for the conjunction in

order to prevent the reachability of the inconsistent state \perp .

Compatibility: This issue is directly borrowed from Interface Automata and relies on a notion of *illegal* state, as pointed out in Section VI-A. Given two Modal Interfaces \mathcal{C}_1 and \mathcal{C}_2 , a state (s_1, s_2) of the parallel product $\mathcal{C} = \mathcal{C}_1 \otimes \mathcal{C}_2$ is said to be illegal if \mathcal{C}_1 may offer from s_1 an output action that may not be accepted as an input by \mathcal{C}_2 from s_2 , or vice-versa. More formally:

$$\begin{aligned} may_1(s_1, !\sigma) \neq \emptyset \quad \text{and} \quad must_2(s_2, ?\sigma) = \emptyset \\ \text{or} \\ may_2(s_2, !\sigma) \neq \emptyset \quad \text{and} \quad must_1(s_1, ?\sigma) = \emptyset \end{aligned} \quad (24)$$

A *correct* environment for \mathcal{C} then makes illegal states unreachable under sequences of actions controlled by the considered component \mathcal{C} . The maximal behavior of $\mathcal{C} = \mathcal{C}_1 \otimes \mathcal{C}_2$ in such an environment is denoted $\mathcal{C}_1 \parallel \mathcal{C}_2$ and can be obtained from $\mathcal{C}_1 \otimes \mathcal{C}_2$ by pruning away the states that can reach illegal states under sequences of actions controlled by the considered component (see [83]). The modal interfaces \mathcal{C}_1 and \mathcal{C}_2 are then declared *compatible* if applying this pruning does not remove the initial state. The above pruning technique and resulting parallel composition \parallel allows exposing explicitly what are the compatible environments for a product, see the discussion section III-C.

In [13], 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 [13], i.e., two Modal Interfaces are only compatible if incompatibility between two interfaces can occur in any environment. A verification tool called *MIO Workbench* is available.

C. Handling dissimilar alphabets

So far we have discussed relations and operations on Modal Interfaces for the special case in which all involved contracts and implementations possess identical alphabets of events. However, as pointed out in Section V-B, alphabets must be local, not global to the entire system. Consequently, unequal alphabets must be dealt with when handling parallel composition \otimes and conjunction \wedge of contracts.

Contrary to the case of Assume/Guarantee contracts (see Section V-B), using modalities allows for *neutral* procedures for equalizing alphabets in both the conjunction operator \wedge and the composition operator \otimes . The principle is as follows. Observe that, if we use the symbol \rightsquigarrow to denote an arbitrary modality, in Table IV,

$$\begin{aligned} s_1 \xrightarrow{\sigma} s'_1 \quad \text{and} \quad s_2 \rightsquigarrow^{\sigma} \\ \downarrow \\ (s_1, s_2) \rightsquigarrow^{\sigma} \end{aligned} \quad (25)$$

holds if the two interfaces are combined using parallel composition. Similarly, according to Table V,

$$\begin{aligned} s_1 \xrightarrow{\sigma} s'_1 \quad \text{and} \quad s_2 \rightsquigarrow^{\sigma} \\ \downarrow \\ (s_1, s_2) \rightsquigarrow^{\sigma} \end{aligned} \quad (26)$$

holds if the two interfaces are combined using conjunction.

The observation above reveals our solution. To ensure a neutral alphabet extension, we add self-loops labeled with the extra event symbols (those that extend the alphabet) to each state, with *specific* modalities:

- *may* for the case of the conjunction \wedge ;
- *must* for the case of the parallel composition \otimes .

These two types of alphabet extensions were called *weak* and *strong* in [115], see this reference for details. Using these two different kinds of extension, Modal Interfaces instantiate the meta-theory of Table II even in the case of dissimilar alphabets.

In contrast, only the “weak” extension could be defined for Assume/Guarantee contracts, thus explaining the lack of proper handling of alphabet equalization in this framework, see Section V-B.

D. Practical implementation

As usual with testing, unless exhaustive simulations of both M and C can be performed, the methods of double testing (for implementation (22) and refinement (23)) cannot prove $M \models C$ nor $M_1 \leq M_2$, it can only detect violations of it. However, double testing has the advantage of being applicable to any kind of specification and design. This is similar to the use of testers in the implementation of the Assume/Guarantee contracts.

Effective and efficient calculi exist, however, with the due restriction to finite states Modal Interfaces. Considering finite state Modal Interfaces and assuming finiteness of Σ , a design $M = (\Sigma, Q, i^0, \rightarrow)$ implements a Modal Interface $(\Sigma, S, s^0, \text{may}, \text{must})$ if there exists a simulation relation $\rho \subseteq Q \times S$ such that $(i^0, s^0) \in \rho$ and for all $(q, s) \in \rho$, we have:

$$\begin{aligned} \text{if } [q \xrightarrow{\sigma} q'] & \quad \text{then } [\exists s'. s' \in \text{may}(s, \sigma)] \\ \text{if } [\exists s'. s' \in \text{must}(s, \sigma)] & \quad \text{then } [q \xrightarrow{\sigma} q'] \end{aligned}$$

and $(q', s') \in \rho$. Regarding refinement, the definition becomes: $C_1 \leq C_2$ if there exists a simulation relation $\rho \subseteq S_1 \times S_2$ such that $(s_1^0, s_2^0) \in \rho$ and for all $(s_1, s_2) \in \rho$, we have:

$$\begin{aligned} \text{if } [\exists s'_1. s'_1 \in \text{may}(s_1, \sigma)] & \quad \text{then } [\exists s'_2. s'_2 \in \text{may}(s_2, \sigma)] \\ \text{if } [\exists s'_2. s'_2 \in \text{must}(s_2, \sigma)] & \quad \text{then } [\exists s'_1. s'_1 \in \text{must}(s_1, \sigma)] \end{aligned}$$

and $(s'_1, s'_2) \in \rho$.

Determinism plays a role in the modal theory. A Modal Interface is said to be deterministic if its *may*-transition relation is deterministic, meaning that for all state $s \in S$ and $\sigma \in \Sigma$, $|\text{may}(s, \sigma)| \leq 1$. For nondeterministic Modal Interfaces, modal refinement is *incomplete* [93]: there are nondeterministic Modal Interfaces C_1 and C_2 for which the set of implementations of C_1 is included in that of C_2 without C_1 being a modal refinement of C_2 . Hence operator (3) from Table II in Section III-B is not exactly instantiated but only approximated in a sound way. A decision procedure for model inclusion of nondeterministic Modal Interfaces does exist but it turns out to be EXPTIME-complete [7], [14] whereas the

problem is PTIME-complete if determinism is assumed [115], [15]. The benefits of the determinism assumption in terms of complexity for various decision problems on modal specifications is underlined in [15].

E. Expressing contracts with assumptions and guarantees as Modal Interfaces

The expressivity of Modal Specifications has been characterized as a strict fragment of the Hennessy-Milner logic in [30] and also as a strict fragment of the mu-calculus in [66]. The formalism is rich enough to specify safety properties as well as restricted forms of liveness properties.

One advantage of the theory of Assume/Guarantee contracts is the flexibility offered by explicitly splitting assumptions from guarantees. This advantage is preserved by Modal Interfaces, since a natural connection between a pair (A, G) and its associated Modal Interface C exists. Such a connection can be established in two different ways.

In a first approach, we can regard a Modal Interface as being defined by its table of transition rules: while in state s , action a *may* or *must* result in a move to state s' . If action a is labeled by a “?”, expressing that it is under the control of the environment, then this rule is interpreted as an assumption. Conversely, if action a is labeled by a “!”, expressing that it is under the control of the component, then this rule is seen as a guarantee. Observe that this coding also encompasses strong/weak assumptions. If an action $a?$ is not offered by the interface in some state s , then this corresponds to a strong assumption on the environment, as the latter cannot submit action a while the component is in state s . To make the refusal of $a?$ in state s a weak assumption, one can proceed as follows: action $a?$ is allowed but leads to a special state \top in which every action is allowed but nothing is guaranteed by the component.

In an alternative approach, consider the contract formed by a pair of Modal Specifications C_A, C_G representing respectively its assumptions and its guarantees. A design M for this contract is correct if, when put under any environment meeting all specified assumptions, it satisfies the entailed guarantees. More formally,

$$\begin{aligned} M \models (C_A, C_G) & \text{ if and only if,} \\ \text{for any environment } E, & E \models C_A \Rightarrow M \times E \models C_G. \end{aligned} \quad (27)$$

According to Equation (27), these correct designs are exactly captured by the quotient C_G/C_A . Moreover, if the initial contract is refined, that is, following the previous paragraph on quotient, $C'_G \leq C_G$ and $C'_A \leq C_A$ then the monolithic version of the contract obtained by quotient is also refined, that is $C'_G/C'_A \leq C_G/C_A$.

A contract theory based on assume/guarantee contracts viewed as pairs of Modal Specifications is developed in [70]. 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.

Last, the synthesis of Modal Interfaces from higher-level specifications has been studied for the case of scenarios. In [120], Existential Live Sequence Charts are translated into Modal Specifications, hence providing a mean to specify modal contracts.

F. Refinements and Extensions

Assume/Guarantee contracts were born with the notion of time built-in, since they were originally proposed as a hybrid model, and in fact the Contract Specification Language (CSL) proposed in SPEEDS has constructs that support the specification of time. Handling synchronicity is also built-in for Assume/Guarantee contracts. Of course, all this requires using tests for the practical implementation as indicated in Section V-E.

For Modal Interfaces is not so; however, it is possible to extend Modal Interfaces to deal with time, synchronicity and resources in general. We review these and other useful extensions to Modal Interfaces.

1) *A refinement of Modal Interfaces: Acceptance Interfaces:* Acceptance Interfaces were proposed by J-B. Raclet [113], [114]. We present them here because they go one step further beyond Modal Interfaces in capturing compatibility, see Table II. Informally, an Acceptance Interface consists of a set of states, with, for each state, a set of *ready sets*, where a ready set is a set of possible outgoing transitions from that state. In intuitive terms, an Acceptance Interface explicitly specifies its set of possible implementations.

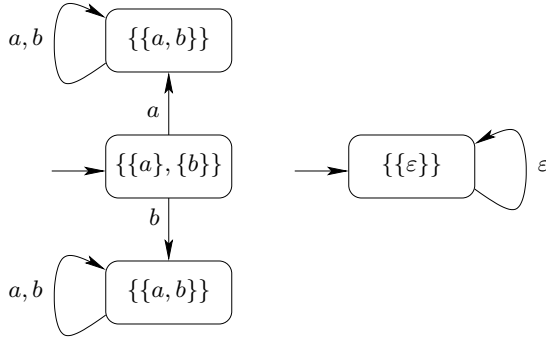


Figure 11. Acceptance Interface. Left: An example. Right: an Acceptance Interface that exactly specifies the obligation for any implementation to be legal.

Figure 11, left, shows an example. In the initial state, a correct implementation can either offer action a as the only possibility to leave the initial state (and then move to the top state), or offer action b as the only possibility to leave the initial state (and then move to the bottom state). In the top or bottom state, any implementation must offer the choice between a or b , and then loop to the same state.¹⁹ Observe

¹⁹This was loosely speaking. A formal presentation should either handle languages or use a simulation relation.

that this set of implementations cannot be specified using a Modal Interface.

A correct implementation for an Acceptance Interface is simply obtained by picking one of the ready sets in each state. Refinement is by inclusion of the sets of ready sets. The definitions of refinement, conjunction, composition, and even quotient exist and are defined very elegantly.

Acceptance Interfaces subsume Modal Interfaces as follows. The set of all *must* transitions from a state form the minimal ready set while the set of all transitions labeled *must* or *may* form the maximal ready set; any ready set sitting between the above two ones is then associated to the considered state.

The most interesting feature of this framework is that it allows for a direct definition of the obligation for any implementation to be legal, see Figure 11, right. The diagram shown is simply the Acceptance Interface having a single state. This state has a set of ready sets reduced to a singleton and consisting of a single self-looping transition labeled with the unobserved event ε , thus enforcing the progress of this specification — having an empty ready set $\{\emptyset\}$ in a state indicates that this state is a trap. Accordingly, any refinement of this Acceptance Interface has the same property, expressing that deadlock or any other kind of blocking is forbidden by any refinement of it. Unfortunately, this mathematical elegance comes at a price. The cost of coding sets of ready sets is prohibitive [113], which makes in the end this framework not practical.

2) *Time:* A first interface theory able to capture the timing aspects of components is Timed Interfaces [54]. Timed Interfaces allows one to specify 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 first timed extension of modal transition systems was published in [36]. 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 [69]. Another timed extension of Modal Specifications was proposed in [22]. In this formalism, transitions are equipped with a modality and a guard on the component clocks, very much like in timed automata. For the subclass of modal event-clock automata, an entire algebra with refinement, conjunction, product, and quotient has been developed in [20], [21].

The timed interface theory proposed in [48] fills a gap in the work introduced in [54] 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 [49].

3) *Synchronicity:* 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 [39]. In Moore Interfaces, each variable is either an input or an output, and this status does not change in time. Bidirectional Interfaces offer added flexibility by allowing variables to change I/O status, depending on the local state of the interface. Communication by shared variable is thus supported and, for instance, allows distributed protocols or shared buses to be modeled. In both formalisms, two interfaces are deemed compatible whenever no variable is an output of both interfaces at the same time, and every legal valuation of the output variables of one interface satisfies the input predicate of the other. The main result of the paper is that parallel composition of compatible interfaces is monotonic with respect to refinement. Note that Moore and Bidirectional Interfaces force a delay of at least one transition between causally dependent input and output variables, exactly like Moore machines. In [61], the framework of synchronous interfaces is enriched with a notion of conjunction (called *shared refinement*).

Sociable interfaces [51] combine the approach presented in the previous paragraph with interface automata [52], [53] by enabling communication via shared variables and actions²⁰. First, the same action can appear as a label of both input and output transitions. Secondly, global variables do not belong to any specific interface and can thus be updated by multiple interfaces. Consequently, communication and synchronization can be one-to-one, one-to-many, many-to-one, and many-to-many. Symbolic algorithms for checking the compatibility and refinement of sociable interfaces have been implemented in TICC [3].

In Moore interfaces, legal values of the input variables and consequent legal values of the output variables are not causally related. Relational Interfaces [123] 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 formulae over the input and output variables. Two types of composition are then considered, connection and feedback. Given two relational interfaces C_1 and C_2 , the first one consists in connecting some of the output variables of C_1 to some of the input variables of C_2 whereas feedback composition allows one to connect an output variable of an interface to one of its own inputs. The developed theory supports refinement, compatibility and also conjunction.

4) *Resources*: *Resource Interfaces* [40] can be used to enrich a variety of interface formalisms (Interface Automata [52], Assume/Guarantee Interfaces [53], ...) 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

²⁰This formalism is thus not purely synchronous and is mentioned in this section with a slight abuse.

and rewards are explicit rather than being defined implicitly as solutions of numerical constraints, this formalism does not allow one to reason about the variability of resource consumption across a set of logically correct implementations.

5) *Stochastic*: Exactly like the Interval Markov Chain (IMC) formalism [85] they generalize, Constraint Markov Chains (CMC) [32] are abstractions of a (possibly infinite) sets of Discrete Time Markov Chains. Instead of assigning a fixed probability to each transition, transition probabilities are kept symbolic and defined as solutions of a set of first order formulas. Variability across implementations is made possible not only with symbolic transition probabilities, but also thanks to the labelling of each state by a set of valuations or sets of atomic propositions. This allows CMCs to be composed thanks to a conjunction and a product operators. While the existence of a residuation operator remains an open problem, CMCs form an interface theory in which satisfaction and refinement are decidable, and compositions can be computed using quantifier elimination algorithms. In particular, CMCs with polynomial constraints form the least class of CMCs closed under all composition operators.

Abstract Probabilistic Automata (APA) [58] 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.

G. Discussion

To conclude, Modal Interfaces offer the best solution for a theory of contracts, from the theoretical point of view. While not direct, the link to Assumptions and Guarantees is simple and clear enough. Modal Interfaces are the only framework meeting all the requirements of Table II on the meta-theory at a reasonable computational cost. Provided that a Modal Interface is described as a set of transition rules, splitting it into Assumptions and Guarantees is easy. In addition, by offering the *must* modality, Modal Interfaces allow for specifying a large class of obligations, not just safety properties. To keep within decidable and effective classes of models, Modal Interfaces must come with restrictions: state spaces must be finite, event and action alphabets must be finite. Timed and even probabilistic extensions exist, but are sophisticated. To overcome these limitations, Modal Interfaces were presented with no such restriction. In turn, a test-based approach to handle them was proposed, which offers the usual generality of test-based methods and the same restrictions in terms of completeness. This approach also has the merit of not requiring any costly tool but simulation. To the best of our knowledge, this is the first time a test-based approach is proposed for modal models.

VII. CONCLUDING REMARKS

We are just at the beginning of a long and winding road towards the development of full fledged design methodologies,

models, tools and flows based on contracts. In this section we discuss the status of contract-based design, both from a theoretical point of view and with regard to applications and use. We first review where we are. Then we discuss where we should go.

Where are we?

From a birds eye view, requirements on contract or interface theories can be summarized as follows. Such theories should support distributed and concurrent development of complex systems across supplier chains, allowing for reuse and variability. A central concern is the safe and error-free integration of independent sub-system or component developments and the reuse of analyses for incremental certification. Independent development of sub-systems is thus essential. And so is the orthogonalization of the development of the different viewpoints on the system, for development by different teams using different frameworks. As for the question raised in the section title, the answer differs, depending on the perspective.

Theory: The understanding of the theory of interfaces or contracts has made significant progress since it started as a recognized topic about year 2000. Three tracks have been followed. Interface Automata came first by pushing in the forefront the issue of compatibility when composing components. Also, alternate simulation having a game theoretic flavor — the component plays against the environment — gave raise to the concept of refinement. Extensions were developed (for time, resources, and shared variable models for software). Later on, Assume/Guarantee contracts were proposed by researchers from SPEEDS. Assumption and Guarantees came in the forefront; the need for handling the conjunction (in addition to the parallel composition) emerged. In parallel, the inventors of modal automata and specifications realized they could mimic interfaces with their own framework. With the subsequent handling of compatibility, Modal Interfaces appeared as the ultimate solution. Extensions have been proposed; more are under development.

While preparing this paper, we realized that test-based and abstraction-based approaches have important practical implications, whereas theory experts did not really pay attention to these approaches. These two approaches can accommodate data of arbitrary type, and even continuous time dynamics — like in the analog design example of companion paper [16]. In addition, test based approaches can offer basic services with minimal (if any) specific tooling. Indeed, they are a bridge towards contract-based design. These findings motivated our decision to include observers as part of relevant techniques for contract-based design. In addition we decided to explain, for each framework, what can be done using tests only. Clearly, the meta-theory with all its concepts cannot be implemented as is. Surprisingly, however, a significant part of it can as shown in this paper. This aspect is thus an original and useful contribution. We also believe that test based techniques for contracts deserve further effort and study.

Methodology: Its theoretical foundations are certainly not enough to gain acceptance of contract-based design by engineers, albeit they are necessary to build a solid set of models, tools and flows that will be relevant in applications. Simple selling arguments are the following: Using contracts helps identify responsibilities by distinguishing the obligations of the considered sub-system (the “guarantees”) from its conditions for use (the “assumptions”). It also enables orthogonal handling of sub-systems and viewpoints and it contributes to explaining and tracing the impact of changes performed on a sub-system. Contracts can refer to the architecture and the layering of the design. This is why we considered “horizontal” and “vertical” contracts.

Yet, we must acknowledge that no commercial offering exists today that supports theory in combination with concepts that speak well to the engineer. Perhaps the most advanced development in this direction is the one that started with the SPEEDS project and is now pursued within the CESAR project. A comprehensive tool chain is being developed that includes: a contract editor offering a library of patterns similar to PSL statements, possibly enhanced for timing or safety analysis; the HRC meta-model (Heterogeneous Rich Component) supporting the needed entities to handle contracts; compositional means checking the correctness of implementations for functional, timing and safety viewpoints against contracts. This is achieved using sound circular reasoning with a combination of model checking or simulation-based testing technology.

Where should we go?

The status and evolution of the theory suggests that a new set of tools and methods could be built on top of the Modal Interface family of theories to develop a contract-based flow as follows: Modal Interfaces can be specified as a table of transition rules relating the current state or location, the event being emitted or the action being performed, and the next state or location. When doing so, the splitting of the table into assumptions and guarantees becomes clear.

How these tables should be expressed depends on the considered stage of the design process. For requirements capture, relevant formalisms include: textual syntax à la PSL, guided natural language (often referred to as “boiler plates”), or even graphical notations such as LiveSequenceCharts. These can be enhanced with timing or safety features to address the corresponding viewpoints.

For the subsequent stage of detailed design, natural candidate tools could be dialects on top of environments such as Simulink and Scade, namely a combination of data-flow diagrams and hierarchical automata or statecharts.

Having contracts specified, the next issue is how to manage them: providing support to check contract consistency and compatibility; combining viewpoints, composing contracts from different sub-systems; checking refinement; providing support for the budgeting of sub-contracts to suppliers; checking the correctness of an implementation against a contract. These different tasks can be performed either using a combination of simulation and testing or observers (thus getting

semi-decisions) or using analysis. Both ways have their virtues and should be considered.

While the Modal Interface approach is appealing since it is based on a robust theory, it is by no means the only one that should be considered. In particular, solutions to the issues detailed above for contract management can be and should be considered in any design flow as described in [16]. We are aware of a number of tools that are being developed to specify and manage contracts in addition to the ones being built in CESAR. These developments may very well contribute to another cornerstone of a rigorous system level design methodology to make it a reality.

REFERENCES

- [1] Martin Abadi and Luca Cardelli. *A Theory of Objects*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1st edition, 1996.
- [2] Yael Abarbanel, Ilan Beer, Leonid Gluhovsky, Sharon Keidar, and Yaron Wolfsthal. FoCs - Automatic Generation of Simulation Checkers from Formal Specifications. In E. Emerson and A. Sistla, editors, *Computer Aided Verification*, volume 1855 of *Lecture Notes in Computer Science*, pages 538–542. Springer Berlin / Heidelberg, 2000.
- [3] B. Thomas Adler, Luca de Alfaro, Leandro Dias da Silva, Marco Faella, Axel Legay, Vishwanath Raman, and Pritam Roy. Ticc: A Tool for Interface Compatibility and Composition. In *Proc. of the 18th International Conference on Computer Aided Verification (CAV'06)*, volume 4144 of *Lecture Notes in Computer Science*, pages 59–62. Springer, 2006.
- [4] 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.
- [5] Rajeev Alur, Thomas A. Henzinger, and Orna Kupferman. Alternating-time temporal logic. *J. ACM*, 49(5):672–713, 2002.
- [6] 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.
- [7] 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.
- [8] Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking*. MIT Press, Cambridge, 2008.
- [9] 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.
- [10] 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.
- [11] Felice Balarin, Yosinori Watanabe, Harry Hsieh, Luciano Lavagno, Claudio Passerone, and Alberto L. Sangiovanni-Vincentelli. Metropolis: An Integrated Electronic System Design Environment. *IEEE Computer*, 36(4):45–52, 2003.
- [12] 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.
- [13] 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.
- [14] 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.
- [15] Nikola Benes, Jan Kretínský, Kim Guldstrand Larsen, and Jirí Srba. On determinism in modal transition systems. *Theoretical Computer Science*, 410(41):4026–4043, 2009.
- [16] A. Benveniste, W. Damm, A. Sangiovanni-Vincentelli, D. Nickovic, P. Reinkemeier, and R. Passerone. Contracts for the Design of Embedded Systems Part I: Methodology and Use Cases. this issue.
- [17] Albert Benveniste and Gérard Berry. The synchronous approach to reactive realtime systems. *Proceedings of the IEEE*, 79(9):1270–1282, 1991.
- [18] 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.
- [19] 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.
- [20] 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.
- [21] 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.
- [22] 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.
- [23] Purandar Bhaduri and S. Ramesh. Interface synthesis and protocol conversion. *Formal Aspects of Computing*, 20(2):205–224, 2008.
- [24] 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.
- [25] Simon Bliudze and Joseph Sifakis. The Algebra of Connectors - Structuring Interaction in BIP. *IEEE Trans. Computers*, 57(10):1315–1330, 2008.
- [26] R. Bloem and B. Jobstmann. Manual for property-based synthesis tool. Technical Report Prosyd D2.2/3, 2006.
- [27] 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.
- [28] Jürgen Bohn, Werner Damm, Hartmut Wittke, Jochen Klose, and Adam Moik. Modeling and validating train system applications using statemate and live sequence charts. In *Proceedings of the Conference on Integrated Design and Process Technology (IDPT2002)*, Society for Design and Process Science (2002, 2002).
- [29] 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.
- [30] Gérard Boudol and Kim Guldstrand Larsen. Graphical versus logical specifications. *Theor. Comput. Sci.*, 106(1):3–20, 1992.
- [31] Ed Brinksma and Jan Tretmans. Testing transition systems: An annotated bibliography. In Cassez et al. [35], pages 187–195.
- [32] 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.
- [33] 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.
- [34] C. G. Cassandras and Lafortune S. *Introduction to Discrete Event Systems — Second Edition*. Springer, 2008.
- [35] Franck Cassez, Claude Jard, Brigitte Rozoy, and Mark Dermot Ryan, editors. *Modeling and Verification of Parallel Processes, 4th Summer School, MOVEP 2000, Nantes, France, June 19-23, 2000*, volume 2067 of *Lecture Notes in Computer Science*. Springer, 2001.

- [36] 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.
- [37] Arindam Chakrabarti. *A Framework for Compositional Design and Analysis of Systems*. PhD thesis, EECS Department, University of California, Berkeley, Dec 2007.
- [38] Arindam Chakrabarti, Luca de Alfaro, Thomas A. Henzinger, Marcin Jurdzinski, and Freddy Y. C. Mang. Interface compatibility checking for software modules. In Ed Brinksma and Kim Guldstrand Larsen, editors, *CAV*, volume 2404 of *Lecture Notes in Computer Science*, pages 428–441. Springer, 2002.
- [39] 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.
- [40] 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.
- [41] 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.
- [42] E. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 1999.
- [43] 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.
- [44] Werner Damm and David Harel. LSCs: Breathing Life into Message Sequence Charts. *Formal Methods in System Design*, 19(1):45–80, 2001.
- [45] Werner Damm, Tobe Toben, and Bernd Westphal. On the expressive power of live sequence charts. In Thomas Reps, Mooly Sagiv, and Jörg Bauer, editors, *Program Analysis and Compilation, Theory and Practice: Essays Dedicated to Reinhard Wilhelm*, volume 4444 of *Lecture Notes in Computer Science*, pages 225–246. Springer-Verlag, May 2007.
- [46] Werner Damm and Bernd Westphal. Live and let die: LSC based verification of UML models. *Sci. Comput. Program.*, 55(1-3):117–159, 2005.
- [47] Werner Damm and Bernd Westphal. Live and Let Die: LSC-based Verification of UML-Models. *Science of Computer Programming*, 55(1-3):117–159, March 2005.
- [48] 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.
- [49] 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.
- [50] 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.
- [51] 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.
- [52] 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.
- [53] 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.
- [54] 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.
- [55] Benoît Delahaye. *Modular Specification and Compositional Analysis of Stochastic Systems*. PhD thesis, Université de Rennes 1, 2010.
- [56] 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.
- [57] 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.
- [58] 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.
- [59] Douglas Densmore, Sanjay Rekh, and Alberto L. Sangiovanni-Vincentelli. Microarchitecture development via metropolis successive platform refinement. In *DATE*, pages 346–351. IEEE Computer Society, 2004.
- [60] 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.
- [61] 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.
- [62] Dumitru Potop-Butucaru and Stephen Edwards and Gérard Berry. *Compiling Esterel*. Springer V., 2007. ISBN: 0387706267.
- [63] 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.
- [64] 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 accelera lrm january 2003. Technical report, IBM, 2003.
- [65] Cindy Eisner, Dana Fisman, John Havlicek, Yoav 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.
- [66] G. Feuillade. Modal specifications are a syntactic fragment of the mu-calculus. Research Report RR-5612, INRIA, June 2005.
- [67] 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.
- [68] Frédéric Boussinot and Robert de Simone. The Esterel language. *Proceedings of the IEEE*, 79(9):1293–1304, 1991.
- [69] 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.
- [70] G. Gössler and J.-B. Ralet. 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.
- [71] 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.
- [72] 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.
- [73] Nicolas Halbwachs. *Synchronous programming of reactive systems*. Kluwer Academic Pub., 1993.
- [74] 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.

- [75] 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.
- [76] 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.
- [77] David Harel, Hillel Kugler, Shahar Maoz, and Itai Segall. Accelerating smart play-out. In Jan van Leeuwen, Anca Muscholl, David Peleg, Jaroslav Pokorný, and Bernhard Rumpe, editors, *SOFSEM*, volume 5901 of *Lecture Notes in Computer Science*, pages 477–488. Springer, 2010.
- [78] David Harel, Shahar Maoz, Smadar Szekely, and Daniel Barkan. Playgo: towards a comprehensive tool for scenario based programming. In Charles Pecheur, Jamie Andrews, and Elisabetta Di Nitto, editors, *ASE*, pages 359–360. ACM, 2010.
- [79] 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>.
- [80] Jifeng He and C. A. R. Hoare. Unifying theories of programming. In *4th International Seminar on Relational Methods in Logic, Algebra and Computer Science (RelMiCS'98)*, pages 97–99, 1998.
- [81] Leszek Holenderski. Compositional verification of synchronous networks. In Mathai Joseph, editor, *FTRFTT*, volume 1926 of *Lecture Notes in Computer Science*, pages 214–227. Springer, 2000.
- [82] INCOSE. Incoese systems engineering handbook, 2010. <http://www.incose.org/ProductsPubs/products/sehandbook.aspx>.
- [83] J-B. Racllet and E. Badouel and A. Benveniste and B. Caillaud and Axel Legay and R. Passerone. Modal Interfaces: Unifying Interface Automata and Modal Specifications. In Lee, I. and Sifakis, J., editor, *Proc. of the 9th International Conference on Embedded Software (EMSOFT'09)*. ACM Press, 2009.
- [84] Thierry Jéron and Pierre Morel. Test generation derived from model-checking. In Nicolas Halbwachs and Doron Peled, editors, *CAV*, volume 1633 of *Lecture Notes in Computer Science*, pages 108–121. Springer, 1999.
- [85] B. Jonsson and K. G. Larsen. Specification and refinement of probabilistic processes. In *Logic in Computer Science (LICS)*, pages 266–277. IEEE Computer, 1991.
- [86] Gabor Karsai, Janos Sztipanovitz, Akos Ledecz, and Ted Bapty. Model-integrated development of embedded software. *Proceedings of the IEEE*, 91(1), January 2003.
- [87] Jochen Klose. *Live Sequence Charts: A Graphical Formalism for the Specification of Communication Behavior*. PhD thesis, Carl von Ossietzky University of Oldenburg, 2003.
- [88] Leslie Lamport. Proving the correctness of multiprocess programs. *IEEE Trans. Software Eng.*, 3(2):125–143, 1977.
- [89] 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.
- [90] 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.
- [91] 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.
- [92] 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.
- [93] 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.
- [94] 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.
- [95] 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.
- [96] A. Ledecz, 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.
- [97] Akos Ledecz, 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.
- [98] D. Lee and M. Yannakakis. Principles and methods of testing finite state machines - a survey. *Proceedings of the IEEE*, 84(8):1090–1123, August 1996.
- [99] Axel Legay, Luca de Alfaro, and Marco Faella. An Introduction to the Tool Ticc. In Serge Autexier, Stephan Merz, Leendert W. N. van der Torre, Reinhard Wilhelm, and Pierre Wolper, editors, *Trustworthy Software*, volume 3 of *OASICS*. Internationales Begegnungs- und Forschungszentrum fuer Informatik (IBFI), Schloss Dagstuhl, Germany, 2006.
- [100] Gerald Lüttgen and Walter Vogler. Safe reasoning with logic lts. *Theoretical Computer Science*, 2011. to appear.
- [101] Nancy Lynch and Mark R. Tuttle. An introduction to Input/Output automata. *CWI-quarterly*, 2(3), 1989.
- [102] Zohar Manna and Amir Pnueli. *Temporal verification of reactive systems: Safety*. Springer, 1995.
- [103] 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.
- [104] 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.
- [105] B. Meyer. Applying “design by contract”. *IEEE Computer*, 25(10):40–51, October 1992.
- [106] 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.
- [107] Object constraint language, version 2.0. OMG Available Specification formal/06-05-01, Object Management Group, May 2006.
- [108] The Design Automation Standards Committee of the IEEE Computer Society, editor. *1850-2010 - IEEE Standard for Property Specification Language (PSL)*. IEEE Computer Society, 2010.
- [109] Paul Le Guernic and Thiserry Gautier and Michel Le Borgne and Claude Le Maire. Programming real-time applications with Signal. *Proceedings of the IEEE*, 79(9):1321–1336, 1991.
- [110] I. Pill, B. Jobstmann, R. Bloem, R. Frank, M. Moulin, B. Sterin, M. Roveri, and S. Semprini. Property simulation. Technical Report ProsyD1.2/1, 2005.
- [111] Amir Pnueli. The temporal logic of programs. In *FOCS*, pages 46–57. IEEE, 1977.
- [112] Terry Quatrani. *Visual modeling with Rational Rose 2000 and UML (2nd ed.)*. Addison-Wesley Longman Ltd., Essex, UK, UK, 2000.
- [113] Jean-Baptiste Racllet. *Quotient de spécifications pour la réutilisation de composants*. PhD thesis, Ecole doctorale Matisse, université de Rennes I, November 2007.
- [114] Jean-Baptiste Racllet. Residual for component specifications. In *Proc. of the 4th International Workshop on Formal Aspects of Component Software (FACS'07)*, 2007.
- [115] Jean-Baptiste Racllet, Eric Badouel, Albert Benveniste, Benoit 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.
- [116] John C. Reynolds. *Theories of Programming Languages*. Cambridge University Press, 1998.
- [117] 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.
- [118] Ingo Schinz, Tobe Toben, Christian Mrugalla, and Bernd Westphal. The rhapsody uml verification environment. In Jorge R. Cuellar and Zhiming Liu, editors, *Proceedings of the 2nd International Conference on Software Engineering and Formal Methods (SEFM 2004)*, Beijing, China, pages 174–183. IEEE, September 2004.

- [119] D. Schmidt. Model-driven engineering. *IEEE Computer*, pages 25–31, February 2006.
- [120] German Sibay, Sebastian Uchitel, and Víctor Braberman. Existential live sequence charts revisited,. In *ICSE 2008: 30th International Conference on Software Engineering*. ACM, May 2008.
- [121] 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.
- [122] Jan Tretmans. Test generation with inputs, outputs and repetitive quiescence. *Software - Concepts and Tools*, 17(3):103–120, 1996.
- [123] 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.
- [124] 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.
- [125] 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.
- [126] Martin Westhead and Simin Nadjm-Tehrani. Verification of embedded systems using synchronous observers. In Bengt Jonsson and Joachim Parrow, editors, *FTRTFT*, volume 1135 of *Lecture Notes in Computer Science*, pages 405–419. Springer, 1996.
- [127] Bernd Westphal. Lsc verification for uml models with unbounded creation and destruction. In Byron Cook, Willem Visser, and Scott Stoller, editors, *Proceedings of the Workshop on Software Model Checking (SoftMC 2005)*, volume 144 of *ENTCS*, pages 133–145. Elsevier B.V., July 2005.