From Interface Automata to Hypercontracts*

Inigo Incer[1], Albert Benveniste[2], Alberto Sangiovanni-Vincentelli[1], and Sanjit A. Seshia[1]

[1] University of California, Berkeley, USA
[2] INRIA/IRISA, Rennes, France

Abstract. de Alfaro and Henzinger's interface automata brought renewed vigor to the tasks of specifying software formally and reasoning about systems compositionally. The key ingredients to this approach were the separation of concerns between environment and implementation, a light-weight behavioral interface that enabled more comprehensive compatibility checks than those allowed by type checking, and the notion of *optimistic* composition of specifications. This new impetus helped launch a research program of formally analyzing and designing general cyber-physical systems compositionally, where contract-based design has been playing a fundamental role. In this paper, we discuss the path connecting interface automata to the theory of contracts, with a special emphasis on hypercontracts, a recent development of contract theory.

1 Introduction

The task of formally verifying software was first enunciated by Turing in 1949. His three-page document starts with a familiar paradigm: "How can one check a routine in the sense of making sure that it is right? In order that the man who checks may not have too difficult a task the programmer should make a number of definite assertions which can be checked individually, and from which the correctness of the whole programme easily follows" [36]. The software verification agenda gained force and focus some twenty years later with Floyd [15] and Hoare [19]. An important part of the verification effort is writing the specifications that programs should satisfy. A methodology for this task was reported by Parnas in 1972 [28]. Concurrency introduced a new angle to writing specifications: it is no longer sufficient to specify an input/output relation for a concurrently-executing routine, as this maintains an ongoing relation with its environment. This realization led to the definition of a reactive system by Harel and Pnueli in 1985 [17]. Interface automata find a home in this context.

In 1998, de Alfaro and Henzinger introduced Interface Automata (IA), a "light-weight formalism for capturing temporal aspects of software component interfaces which are beyond the reach of traditional type systems" [11]. The formalism used by the authors has several distinguishing traits: the choice of an automata-based language, the partition of symbols into inputs and outputs, a

* This paper is based on [20] and [22].

new refinement relation, the separation of concerns between assumptions from the environment and responsibilities of the object under specification, and an optimistic approach to composing automata.

IA gave rise to a series of *interface theories* which extended it in multiple ways, for example, modal I/O automata [24], resource interfaces [6], timed interfaces [12], timed IO automata [10], permissive interfaces [18], modal interfaces [33], and interfaces with support for component reuse [14]. Moreover, IA influenced approaches to formally design and analyze cyber-physical systems.

From a practical standpoint, a difficulty with interface theories is the tight coupling between assumptions on the environment and guarantees expected from the component. However, the explicit expression of assumptions and guarantees seems to be a conceptually simpler approach to system specification (see [5] chapter 12). Assume-guarantee contract-based design can then be seen as an alternative to formal system specification with a friendlier formalism for practicing designers.

Recognizing the fundamental contributions by Tom Henzinger, this paper is about establishing a relationship between the two approaches and generalizing contracts to a more powerful formalism to extend their applicability.

2 Interface automata

Let Σ be a fixed set of actions[3]. We partition Σ into sets or input and output actions, I and O, respectively.

Definition 1. *An interface automaton* [11] *is a tuple* $A = (I, O, Q, q_0, \rightarrow)$, *where* Q *is a finite set whose elements we call states,* $q_0 \in Q$ *is the initial state, and* $\rightarrow \subseteq Q \times \Sigma \times Q$ *is a deterministic transition relation.*

In the original definition, interface automata (IA) also use the concept of internal actions, which can be represented as output actions in the synchronous case. Syntactically, interface automata are indistinguishable from IO automata [25]. They, however, differ in their associated semantic concepts of refinement and parallel composition. Also, in contrast to IO automata, which require their implementations to be receptive to input actions, IA state the assumptions on the environments in which valid implementations run. In other words, certain moves by the environment may not be allowed by the IA.

Definition 2. *Given two interface automata (IA)* $A_i = (I, O, Q_i, q_{i,0}, \rightarrow_i) \in \mathcal{A}_I$ *for* $i \in \{1, 2\}$, *the state* $q_1 \in Q_1$ *refines* $q_2 \in Q_2$, *written* $q_1 \leq q_2$, *if*

- $\forall \sigma \in O, q_1' \in Q_1.\ q_1 \xrightarrow{\sigma}_1 q_1' \Rightarrow \exists q_2' \in Q_2.\ q_2 \xrightarrow{\sigma}_2 q_2'$ *and* $q_1' \leq q_2'$ *and*
- $\forall \sigma \in I, q_2' \in Q_2.\ q_2 \xrightarrow{\sigma}_2 q_2' \Rightarrow \exists q_1' \in Q_1.\ q_1 \xrightarrow{\sigma}_1 q_1'$ *and* $q_1' \leq q_2'$.

A_1 *refines* A_2, *written* $A_1 \leq A_2$, *if* $q_{1,0} \leq q_{2,0}$.

[3] To simplify our development, we assume that all interface automata share the same action alphabet.

Note that this is the definition of alternating simulation by Alur et al. [2]. Compared to the usual definition of simulation for automata, alternating simulation builds on a game view of automata, in which inputs are seen as adversarial.

IA are composed according to an optimistic approach: for two IA to be composed, it suffices that there be an environment that allows both to operate.

Definition 3. *Let $A_1 = (I_1, O_1, Q_1, q_{1,0}, \to_1)$ and $A_2 = (I_2, O_2, Q_2, q_{2,0}, \to_2)$ be two IA. Let $I_1 \cup I_2 = \Sigma$[4]. Then, the composition of A_1 and A_2, $A_1 \parallel A_2$, is an IA $(I, O, Q, (q_{1,0}, q_{2,0}), \to_c)$, where $I = I_1 \cap I_2$, $O = O_1 \cup O_2$, and the set of states and the transition relation are obtained as follows:*

– *Initialize $Q \coloneqq Q_1 \times Q_2$. For every $\sigma \in \Sigma$, $(q_1, q_2) \xrightarrow{\sigma}_c (q_1', q_2')$ if $q_1 \xrightarrow{\sigma}_1 q_1'$ and $q_2 \xrightarrow{\sigma}_2 q_2'$.*
– *Initialize the set of invalid states to those states where one interface automaton can generate an output action that the other interface automaton does not accept:*

$$N \coloneqq \left\{ (q_1, q_2) \in Q_1 \times Q_2 \;\middle|\; \begin{array}{l} \exists\, q_2' \in Q_2, \sigma \in O_2\, \forall q_1' \in Q_1.\; q_2 \xrightarrow{\sigma}_2 q_2' \wedge \neg \left(q_1 \xrightarrow{\sigma}_1 q_1' \right) \text{ or} \\ \exists\, q_1' \in Q_1, \sigma \in O_1\, \forall q_2' \in Q_2.\; q_1 \xrightarrow{\sigma}_1 q_1' \wedge \neg \left(q_2 \xrightarrow{\sigma}_2 q_2' \right) \end{array} \right\}.$$

– *Also consider invalid any state in which an output action of one of the interface automata makes a transition to an invalid state, i.e., iterate this rule until convergence:*

$$N \coloneqq N \cup \left\{ (q_1, q_2) \in Q_1 \times Q_2 \;\middle|\; \begin{array}{l} \exists\, (q_1', q_2') \in N, \sigma \in O_1 \cup O_2. \\ (q_1, q_2) \xrightarrow{\sigma}_c (q_1', q_2') \end{array} \right\}.$$

– *Remove the invalid states from the IA to obtain*

$$\begin{aligned} Q &\coloneqq Q \setminus N \text{ and} \\ \to_c &\coloneqq \to_c \setminus \{(q, \sigma, q') \in \to_c \mid q \in N \text{ or } q' \in N\}. \end{aligned}$$

Abadi and Lamport [1] studied the composition of specifications split into assumptions on the environment and responsibilities of the object under specification. As shown by Benveniste et al. [5], the composition of interface automata follows the composition principle enunciated by Abadi and Lamport.

3 Cyber-physical systems and assume-guarantee contracts

Cyber-Physical Systems and the practical relevance of traces. The control community had developed since the early 1990's a hybrid system modeling

[4] Since the sets of input and output actions for a given IA partition Σ, this condition is equivalent to requiring that the two IA don't share outputs.

approach based on ODEs plus discrete-time dynamical systems. Solution trajectories of such models are called *traces* in the computer science community. The concept of trace is indeed playing a fundamental role in CPS.

Referring to a paper by Dijkstra from 1965 [13], Lamport writes, "Dijkstra was aware from the beginning of how subtle concurrent algorithms are and how easy it is to get them wrong. He wrote a careful proof of his algorithm. The computational model implicit in his reasoning is that an execution is represented as a sequence of states . . . I have found this to be the most generally useful model of computation—for example, it underlies a Turing machine. I like to call it the standard model" [23]. During the 70s, making statements over the standard model became well established. The trace was the object that enabled us to conclude whether a program satisfied a certain requirement. Traces were given convenient syntactical representations with temporal logic, notably Pnueli's LTL [30] and Clarke and Emerson's CTL [7]. Many algebraic formalisms were also introduced to reason about systems whose components were modeled as collections of traces.

Component-based design. By refocusing on the interfaces of reactive systems, interface automata brought new vigor to research in compositional design. Damm [9] introduced to systems engineering the notion of a *rich component* where functional and non-functional aspects of the design can be captured. This idea motivated the agenda of applying the techniques of formal methods to any kind of CPS, a theory and methodology called contract-based design [4,34]. An agenda for further development of component-based design for software is given in Sifakis's book on rigorous design [35], Section 5.3.

Assumptions and guarantees as first class citizens: Assume-Guarantee contracts. Building on top of trace-based modeling, Assume-Guarantee (AG) contracts [4,5] provide a formal framework of contract-based design, in which assumptions and guarantees are first-class citizens. AG contracts assume that all functional and non-functional behaviors of the system have been modeled in advance. We assume an underlying set $\mathcal{B}$ of behaviors, generalizing traces or executions typical in the computer science literature.

Definition 4. *An assume-guarantee contract is a pair* $\mathcal{C} = (A, G)$*, where* $A \subseteq \mathcal{B}$ *models the assumptions made on the environment, and* $G \subseteq \mathcal{B}$ *the guarantees required of the object under specification.*

An *environment* $E \subseteq \mathcal{B}$ of $\mathcal{C}$ is a component that satisfies the assumptions of the contract, i.e., $E \subseteq A$. An *implementation* $M \subseteq \mathcal{B}$ of $\mathcal{C}$ is a component that satisfies the guarantees of the contract provided that it operates in an environment that satisfies the contract specifications, i.e., if $M \cap E \subseteq G$ for all environments E of $\mathcal{C}$.

Two contracts are *equivalent* if they have the same environments and the same implementations. By observing that the set of implementations is the set of all subsets of $G \cup \neg A$, we deduce that any contract (A, G) has the same environments and implementations as the contract $(A, G \cup \neg A)$. Moreover, we cannot increase the guarantees of this last contract without increasing its implementation set. For this reason, when a contract satisfies the constraint $G = G \cup \neg A$

(which is equivalent to $A \cup G = \mathcal{B}$), the contract is *in canonical form* (some authors call it *saturated form*).

We say that a contract $\mathcal{C}$ *refines* a contract $\mathcal{C}'$ if the environments of $\mathcal{C}'$ are also environments of $\mathcal{C}$ and the implementations of $\mathcal{C}$ are also implementations of $\mathcal{C}'$. That is, the notion of refinement of AG contracts is covariant for implementations and contravariant for environments, just like in interface automata.

Definition 5. *If contracts $\mathcal{C} = (A, G)$ and $\mathcal{C}' = (A', G')$ are in canonical form, we say that $\mathcal{C} \leq \mathcal{C}'$ if $G \subseteq G'$ and $A' \subseteq A$.*

AG contracts are composed according to the composition principle of Abadi and Lamport [1]. When contracts $\mathcal{C}_i$ are composed, this principle states that the composite is the smallest specification satisfying the following constraints:

– any composition of implementations of all $\mathcal{C}_i$ is an implementation of $\mathcal{C}'$; and
– for any $1 \leq j \leq n$, any composition of an environment of $\mathcal{C}'$ with implementations of all $\mathcal{C}_i$ (for $i \neq j$) yields an environment for $\mathcal{C}_j$.

Instantiating this composition for contracts $\mathcal{C}_1 = (A_1, G_1)$ and $\mathcal{C}_2 = (A_2, G_2)$ in canonical form yields

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

It may be the case that the same design element is assigned multiple specifications corresponding to multiple viewpoints, or design concerns [4,29] (e.g., functionality and a performance criterion). Therefore, AG contracts also support an operation of conjunction, which can be used to enforce simultaneously multiple specifications over the same design element. This operation is the least-upper bound with respect to the refinement order of contracts in canonical form. For contracts in canonical form, this operation is

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

In 1997, Negulescu introduced the formalism of process spaces [27]. Process spaces are mathematically the same objects as AG contracts. While Negulescu thought of process spaces are representing specifications for arbitrary software components, AG contracts were introduced to model arbitrary functional and non-functional aspects of CPSs.

3.1 Issues with AG contracts

Algorithmic issues. Core operations for AG contracts require all contracts to be in canonical form. Putting a contract in canonical form is by itself an expensive operation (see, e.g., [5], Chapter 6). In contrast, interface theories in general come with efficient algorithms, making them excellent candidates for internal representations of specifications. Some authors ([5] chapter 10) have therefore proposed to translate contracts expressed as pairs (assumptions, guarantees) into suitable interface models, where the available algorithms are applied. This

approach has the drawback that results cannot be traced back to the original (assumptions, guarantees) formulation.

Trace properties cannot express certain important specifications. AG contracts are capable of expressing any environment and implementation as a set of traces. These are sometimes also called *trace properties* to emphasize the centrality of the trace in their definition. A trace property P can be defined as follows:

$$P = \{b \in \mathcal{B} \mid \phi(b)\},$$

that is, P contains all traces satisfying a certain predicate ϕ. Observe that the traces contained in any subset of P meet predicate ϕ; these subsets thus correspond to components having property P. Safety and liveness properties are trace properties.

However, not all attributes of a system behave in this way. Consider the component with behaviors $M = \{1, 2, 3\}$. Suppose we state the property $P =$ "the average of the behaviors is 2." M clearly satisfies this property, but M has subsets that do not: P is not a trace property. A statement like P is a perfectly reasonable property one may wish to verify in a system. In order to apply assume-guarantee reasoning to properties of this sort, we must extend the theory of contracts.

4 The theory of hypercontracts

4.1 Hyperproperties

The core concept on top of which specifications, interfaces, or contracts can be expressed, is that of *property*. The most basic definition of a property in the formal methods community is "a set of traces." This notion is based on the behavioral approach to system modelling: we assume we start with an underlying set of behaviors $\mathcal{B}$, and properties are defined as subsets of $\mathcal{B}$. In this approach, design elements or components (or environments) are also defined as subsets of $\mathcal{B}$. The difference between properties and components is semantics: a component collects the behaviors that can be observed from that component, while a property collects the behaviors meeting some criterion of interest. We say a component M satisfies a property P, written $M \models P$, when $M \subseteq P$, that is, when the behaviors of M meet the criterion that determines P. Properties of this sort are also called *trace properties*. Many design qualities are of this type, such as *safety*. But there are many system attributes that can only be determined by analyzing multiple traces such as mean response times, reliability, and security attributes. The following running example illustrates this.

Example 1 (Running example). Consider the digital system shown in Figure 1a; this system is similar to those presented in [31,26] to illustrate the noninterference property in security. Here, we have an s-bit secret data input S and an n-bit public input P. The system has an output O. There is also an input H that is equal to zero when the system is being accessed by a user

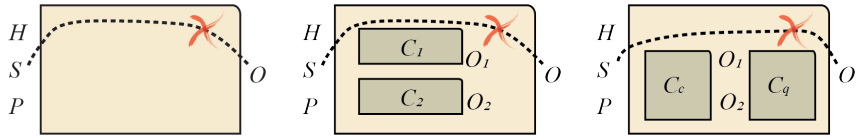

Fig. 1: (a) A digital system with a secret input S and a public input P. The overall system must meet the requirement that the secret input does not affect the value of the output O when the signal H is deasserted (this signal is asserted when a privileged user uses the system). Our agenda for this running example is the following: (b) we will start with two components C_1 and C_2 satisfying respective hypercontracts $\mathcal{C}_1$ and $\mathcal{C}_2$ characterizing information-flow properties of their own; (c) the composition of these two hypercontracts, $\mathcal{C}_c$, will be derived. Through the quotient hypercontract $\mathcal{C}_q$, we will discover the functionality that needs to be added in order for the design to meet the top-level information-flow specification $\mathcal{C}$.

with low-privileges, i.e., a user not allowed to use the secret data, and equal to one otherwise. We wish the overall system to satisfy the property that for all environments with $H = 0$, the implementations can only make the output O depend on P, the public data, not on the secret input S.

A prerequisite for writing this requirement is to be able to express the property that "the output O depends on P, the public data, not on the secret input S". We claim that this property cannot be captured by a trace property. To see this, suppose for simplicity that O, P, and S are 1-bit-long. A trace property that aims at expressing the independence from the secret by making the output O equal to P is

$$P_C = \left\{ \begin{array}{l} (P=1, S=1, O=1), \\ (P=0, S=1, O=0), \\ (P=1, S=0, O=1), \\ (P=0, S=0, O=0) \end{array} \right\}.$$

A valid implementation $M \subseteq P_C$ is the following set of traces

$$M = \left\{ \begin{array}{l} (P=1, S=1, O=1), \\ (P=0, S=0, O=0) \end{array} \right\}.$$

However, the component M leaks the value of S in its output, showing that the independence does not behave as a trace property. Therefore, neither does non-interference.

To overcome this, reformulate the property by simply listing all the subsets of P_C that satisfy the independence requirement, namely:

$$\left\{ \left\{ \begin{array}{l} (P{=}1, S{=}1, O{=}1), \\ (P{=}0, S{=}1, O{=}0), \\ (P{=}1, S{=}0, O{=}1), \\ (P{=}0, S{=}0, O{=}0) \end{array} \right\}, \left\{ \begin{array}{l} (P{=}1, S{=}1, O{=}1), \\ (P{=}1, S{=}0, O{=}1) \end{array} \right\}, \left\{ \begin{array}{l} (P{=}0, S{=}1, O{=}0), \\ (P{=}0, S{=}0, O{=}0) \end{array} \right\} \right\}$$

This rather defines a subset of $2^{\mathcal{B}}$. □

The above discussion suggests the need for a richer formalism for expressing design attributes, namely: Clarkson and Schneider's *hyperproperties* [8], which are precisely subsets of $2^{\mathcal{B}}$. Hyperproperties were anticipated by Raclet's acceptance specifications [32].

Indeed, *non-interference*, introduced by Goguen and Meseguer [16], is a common information-flow attribute. It is a prototypical example of a design quality which trace properties are unable to capture. It can be expressed with hyperproperties, and is in fact one reason behind their introduction. Suppose σ is one of the behaviors that our system can display. The behaviors will be defined as sequences of the states of memory locations through time. For example, if the system has a memory of two bits, the first element of σ will be the initial values of these memory locations; the next element will be the state of these two bits at the next step, and so on. Some of those memory locations we call *privileged*, some *unprivileged*. Let $L_0(\sigma)$ and $L_f(\sigma)$ be the projections of the behavior σ to the unprivileged memory locations of the system, at time zero, and at the final time (when execution is done). We say that a component M meets the non-interference hyperproperty when

$$\forall \sigma, \sigma' \in M.\ L_0(\sigma) = L_0(\sigma') \Rightarrow L_f(\sigma) = L_f(\sigma'),$$

i.e., if two traces begin with the unprivileged locations in the same state, the final state of the unprivileged locations matches. Non-interference is a downward-closed hyperproperty with respect to the subset order [31,26], and a 2-safety hyperproperty—hyperproperties called *k-safety* are those for the refutation of which one must provide at least k traces. In our example, to refute the hyperproperty, it suffices to show two traces that share the same unprivileged initial state, but which differ in the unprivileged final state.

To summarize, many statements of interest are beyond the scope of AG contracts, as AG contracts only support trace properties. Our objective is to develop a theory of assume-guarantee reasoning capable of expressing all attributes of CPSs that have been so far formalized. As hyperproperties have this expressive capability, we focused on an assume-guarantee theory capable of expressing environments and implementations as hyperproperties. We do this in three steps:

1. we consider components coming with notions of preorder (e.g., simulation) and parallel composition;
2. we introduce the notion of a *compset*, a variation of the notion of hyperproperty, equipped with substantial algebraic structure;
3. we build *hypercontracts* as pairs of compsets specifying legal environments and implementations.

4.2 Components

In the theory of hypercontracts, the most primitive concept is the component. Let $(\mathbb{M}, \leq)$ be a preorder. The elements $M \in \mathbb{M}$ are called *components*. We say

that M is a subcomponent of M' when $M \leq M'$. If we represented components as automata, the statement "is a subcomponent of" would correspond to "is simulated by."

There exists a partial binary operation, $\|\colon \mathbb{M}, \mathbb{M} \to \mathbb{M}$, monotonic in both arguments, called *composition*. If $M \parallel M'$ is not defined, we say that M and M' are *non-composable* (and *composable* otherwise). A component E is an environment for component M if E and M are composable. We assume that composition is associative and commutative. We say that a set of components $\{M_i\}_{i=1}^n$ is a decomposition of component M if $M_1 \parallel \ldots \parallel M_n \leq M$.

Example 2 (running example, cont'd). In order to reason about possible decompositions of the system shown in Figure 1a, we introduce the internal variables O_1 and O_2, as shown in Figure 1b. They have lengths o_1 and o_2, respectively. The output O has length o. For simplicity, we will assume that the behaviors of the entire system are stateless (i.e., we don't analyze their change over a notion of a progression or time). In that case, the underlying set of components $\mathbb{M}$ must contain at least the following components:

– For $i \in \{1, 2\}$, components with inputs H, S, P, and output O_i, i.e.,

$$\{(H, S, P, O_1, O_2, O) \mid \exists f \in (2^1 \times 2^s \times 2^n \to 2^{o_i}).\, O_i = f(H, S, P)\}\,.$$

– Components with inputs H, S, P, O_1, O_2, and output O, i.e.,

$$\{(H, S, P, O_1, O_2, O) \mid \exists f \in (2^1 \times 2^s \times 2^n \times 2^{o_1} \times 2^{o_2} \to 2^o).\, O = f(H, S, P, O_1, O_2)\}\,.$$

– Any subset of these components, as these correspond to restricting inputs to subsets of their domains.

In this theory of components, composition is carried out via set intersection. So for example, if for $i \in \{1, 2\}$ we have functions $f_i \in (2^1 \times 2^s \times 2^n \to 2^{o_i})$ and components $M_i = \{(H, S, P, O_1, O_2, O) \mid O_i = f_i(H, S, P)\}$, the composition of these objects is

$$M_1 \parallel M_2 = \left\{ (H, S, P, O_1, O_2, O) \;\middle|\; \begin{matrix} O_1 = f_1(H, S, P) \\ O_2 = f_2(H, S, P) \end{matrix} \right\},$$

which is the set intersection of the components's behaviors. □

4.3 Compsets

CmpSet is a lattice whose objects, called *compsets*, are sets of components. The order of **CmpSet** is set inclusion. In general, not every set of components is an object of **CmpSet**. Compsets boil down to hyperproperties when the underlying component theory represents components as sets of behaviors. Since we assume **CmpSet** is a lattice, the greatest lower bounds and least upper bounds of finite sets are defined. Observe, however, that although the partial order of **CmpSet** is given by the subset order, the meet and join of **CmpSet** are not necessarily

intersection and union, respectively, as the union or intersection of any two elements are not necessarily elements of **CmpSet**. In other words, **CmpSet** is a sublattice of $2^{\mathbb{M}}$.

CmpSet comes with a notion of satisfaction. Suppose $M \in \mathbb{M}$ and H is a compset. We say that M *satisfies* H or conforms to H, written $M \models H$, when $M \in H$. For compsets H, H', we say that H *refines* H', written $H \leq H'$, when $\forall M \models H.\ M \models H'$, i.e., when $H \subseteq H'$.

Example 3 (Running example: non-interference). Regarding the system shown in Figure 1a, we require the top level component to generate the output O independently from the secret input S. We build our theory of compsets by letting the set of elements of **CmpSet** be the set $2^{\mathbb{M}}$. This means that any set of components is a valid compset. The components meeting the top-level non-interference property are those belonging to the compset

$$\{(H, S, P, O_1, O_2, O) \mid \exists f \in (2^1 \times 2^n \to 2^o).\, O = f(H, P)\}\,,$$

i.e., those components for which H and P are sufficient to evaluate O. This corresponds exactly to those components that are insensitive to the secret input S. The join and meet of these compsets is given by set union and intersection, respectively. □

Composition and quotient. Composition in **CmpSet** is element-wise:

$$H \parallel H' = \left\{ M \parallel M' \;\middle|\; \begin{array}{l} M \models H,\ M' \models H', \text{ and} \\ M \text{ and } M' \text{ are composable} \end{array} \right\}. \tag{1}$$

Composition is total and monotonic, i.e., if $H' \leq H''$, then $H \parallel H' \leq H \parallel H''$. It is also commutative and associative, by the commutativity and associativity, respectively, of component composition.

In order to define composition for hypercontracts, we need to assume the existence of a second (but partial) binary operation on the objects of **CmpSet**. This operation is the right adjoint of composition: for compsets H and H', the residual H/H' (also called *quotient*), is given by

$$H/H' = \{M \in \mathbb{M} \mid \{M\} \parallel H' \subseteq H\}\,. \tag{2}$$

The definition of quotient for compsets does not require a notion of quotient for components. However, when such a notion exists, and depending on the structure of **CmpSet**, it can be used to simplify the computation of (2).

Downward-closed compsets. The set of components was introduced with a partial order. We say that a compset H is *downward-closed* when $M' \leq M$ and $M \models H$ imply $M' \models H$, i.e., if a component satisfies a downward-closed compset, so does its subcomponent.

4.4 Hypercontracts

A hypercontract is a specification for a design element that tells what is required from the design element when it operates in an environment that meets the

expectations of the hypercontract. Several ways of specifying hypercontracts can be considered.

Hypercontracts as pairs (environments, closed-system specification). In this setting, a hypercontract is a pair of compsets:

$$\mathcal{C} = (\mathcal{E}, \mathcal{S}) = \text{(environments, closed-system specification)}.$$

$\mathcal{E}$ states the environments in which the object being specified must adhere to the specification. $\mathcal{S}$ states the requirements that the design element must fulfill when operating in an environment which meets the expectations of the hypercontract. We say that a component E *is an environment of hypercontract* $\mathcal{C}$, written $E \models^E \mathcal{C}$, if $E \models \mathcal{E}$. We say that a component M *is an implementation of* $\mathcal{C}$, written $M \models^I \mathcal{C}$, when $M \parallel E \models \mathcal{S}$ for all $E \models \mathcal{E}$. Note that we can use the quotient of compsets to define the set of implementations $\mathcal{I}$ of $\mathcal{C}$ as the compset containing all implementations, i.e., as the quotient:

$$\text{implementations} = \mathcal{I} = \mathcal{S}/\mathcal{E}.$$

A hypercontract with a nonempty set of environments is called *compatible*. If it has a nonempty set of implementations, it is called *consistent*. For $\mathcal{S}$ and $\mathcal{I}$ as above, the compset $\mathcal{E}'$ defined as $\mathcal{E}' = \mathcal{S}/\mathcal{I}$ contains all environments in which the implementations of $\mathcal{C}$ satisfy the specifications of the hypercontract. Thus, we say that a hypercontract is *saturated* if its environments compset is as large as possible in the sense that adding more environments to the hypercontract would reduce its implementations. This means that $\mathcal{C}$ satisfies the following fixpoint equation:

$$\mathcal{E} = \mathcal{S}/\mathcal{I} = \mathcal{S}/(\mathcal{S}/\mathcal{E}).$$

At a first sight, this notion of saturation may seem to go against what for assume-guarantee contracts are called contracts in canonical or saturated form, as we make the definition based on the environments instead of on the implementations. This, however, is a wrong guess: the two definitions for AG contracts and hypercontracts agree. Indeed, for AG contracts, this notion means that the contract $\mathcal{C} = (A, G)$ satisfies $G = G \cup \neg A$. For this AG contract, we can form a hypercontract as follows: if we take the set of environments to be $\mathcal{E} = 2^A$ (i.e., all subsets of A) and the closed system specs to be $\mathcal{S} = 2^G$, we get a hypercontract whose set of implementations is $2^{G \cup \neg A}$, which means that the hypercontract $(2^A, 2^G)$ is saturated.

Hypercontracts as pairs (environments, implementations). Another way to interpret a hypercontract is by telling explicitly which environments and implementations it supports. Thus, we would write the hypercontract as $\mathcal{C} = (\mathcal{E}, \mathcal{I})$. Assume-guarantee theories can differ as to the most convenient representation for their hypercontracts. Moreover, some operations on hypercontracts find their most convenient expression in terms of implementations (e.g., parallel composition), and some in terms of the closed system specifications (e.g., strong merging), as discussed below.

The lattice Contr of hypercontracts. Just as with **CmpSet**, we define **Contr** as a lattice formed by putting together two compsets in one of the above

two ways. Not every pair of compsets is necessarily a valid hypercontract. We will now define the operations that give rise to this lattice.

Preorder. We define a preorder on hypercontracts as follows: we say that $\mathcal{C}$ *refines* $\mathcal{C}'$, written $\mathcal{C} \leq \mathcal{C}'$, when every environment of $\mathcal{C}'$ is an environment of $\mathcal{C}$, and every implementation of $\mathcal{C}$ is an implementation of $\mathcal{C}'$, i.e., $E \models^E \mathcal{C}' \Rightarrow E \models^E \mathcal{C}$ and $M \models^I \mathcal{C} \Rightarrow M \models^I \mathcal{C}'$. Using the notation introduced for compsets, we can express this as

$$\mathcal{E}' \leq \mathcal{E} \text{ and } \mathcal{S}/\mathcal{E} = \mathcal{I} \leq \mathcal{I}' = \mathcal{S}'/\mathcal{E}'.$$

Any two $\mathcal{C}, \mathcal{C}'$ with $\mathcal{C} \leq \mathcal{C}'$ and $\mathcal{C}' \leq \mathcal{C}$ are said to be *equivalent* since they have the same environments and the same implementations. We now obtain some operations using preorders which are defined as the LUB or GLB of **Contr**. We point out that the expressions we obtain are unique up to the preorder, i.e., up to hypercontract equivalence.

GLB and LUB. From the preorder just defined, the GLB of $\mathcal{C}$ and $\mathcal{C}'$ satisfies: $M \models^I \mathcal{C} \wedge \mathcal{C}'$ if and only if $M \models^I \mathcal{C}$ and $M \models^I \mathcal{C}'$; and $E \models^E \mathcal{C} \wedge \mathcal{C}'$ if and only if $E \models^E \mathcal{C}$ or $E \models^E \mathcal{C}'$.

Conversely, the least upper bound satisfies $M \models^I \mathcal{C} \vee \mathcal{C}'$ if and only if $M \models^I \mathcal{C}$ or $M \models^I \mathcal{C}'$, and $E \models^E \mathcal{C} \vee \mathcal{C}'$ if and only if $E \models^E \mathcal{C}$ and $E \models^E \mathcal{C}'$.

The lattice **Contr** has hypercontracts for objects (up to contract equivalence), and meet and join as just described.

Parallel composition. The composition of hypercontracts $\mathcal{C}_i = (\mathcal{E}_i, \mathcal{I}_i)$ for $1 \leq i \leq n$, denoted $\|_i \mathcal{C}_i$, is defined using the Abadi-Lamport composition principle discussed in Section 2:

$$\begin{aligned}\mathcal{C} \parallel \mathcal{C}' &= \bigwedge \left\{ (\mathcal{E}', \mathcal{I}') \in \mathbf{Contr} \;\middle|\; \begin{bmatrix} \mathcal{I}_1 \parallel \ldots \parallel \mathcal{I}_n \leq \mathcal{I}', \text{ and} \\ \mathcal{E}' \parallel \mathcal{I}_1 \parallel \ldots \parallel \hat{\mathcal{I}}_j \parallel \ldots \parallel \mathcal{I}_n \leq \mathcal{E}_j \\ \text{for all } 1 \leq j \leq n \end{bmatrix} \right\} \\ &= \bigwedge \left\{ (\mathcal{E}', \mathcal{I}') \in \mathbf{Contr} \;\middle|\; \begin{bmatrix} \mathcal{I}_1 \parallel \ldots \parallel \mathcal{I}_n \leq \mathcal{I}', \text{ and} \\ \mathcal{E}' \leq \bigwedge_{1 \leq j \leq n} \frac{\mathcal{E}_j}{\mathcal{I}_1 \parallel \ldots \parallel \hat{\mathcal{I}}_j \parallel \ldots \parallel \mathcal{I}_n} \end{bmatrix} \right\}, \end{aligned} \tag{3}$$

where the notation $\hat{\mathcal{I}}_j$ indicates that the composition $\mathcal{I}_1 \parallel \ldots \parallel \hat{\mathcal{I}}_j \parallel \ldots \parallel \mathcal{I}_n$ includes all terms $\mathcal{I}_i$, except for $\mathcal{I}_j$.

Example 4 (Running example, parallel composition). Consider the example shown in Figure 1. We want to state the following requirement for the top-level component: for all environments with $H = 0$, valid implementations can only make the output O depend on P, the public data. We will write a hypercontract for the top-level. We let $\mathcal{C} = (\mathcal{E}, \mathcal{I})$, where

$$\begin{aligned}\mathcal{E} &= \{M \in \mathbb{M} \mid \forall (H, S, P, O_1, O_2, O) \in M.\, H = 0\} \\ \mathcal{I} &= \{M \in \mathbb{M} \mid \exists f \in (2^n \to 2^o). \forall (H, S, P, O_1, O_2, O) \in M.\, H = 0 \to O = f(P)\}.\end{aligned}$$

The environments are all those components only defined for $H = 0$. The implementations are those such that the output is a function of P when $H = 0$.

Let $f^* : 2^n \to 2^o$. Suppose we have two hypercontracts that require their implementations to satisfy the function $O_i = f^*(P)$, one implements it when

$S = 0$, and the other when $S \neq 0$. For simplicity of syntax, let s_1 and s_2 be the propositions $S = 0$ and $S \neq 0$, respectively. Let the two hypercontracts be $\mathcal{C}_i = (\mathcal{E}_i, \mathcal{I}_i)$ for $i \in \{1, 2\}$. We won't place restrictions on the environments for these hypercontracts, so we obtain $\mathcal{E}_i = \mathbb{M}$ and

$$\mathcal{I}_i = \{M \in \mathbb{M} \mid \forall (H, S, P, O_1, O_2, O) \in M. s_i \rightarrow O_i = f^*(P)\}.$$

We now evaluate the composition of these two hypercontracts: $\mathcal{C}_c = \mathcal{C}_1 \parallel \mathcal{C}_2 = (\mathcal{E}_c\,,\, \mathcal{I}_c)$, yielding $\mathcal{E}_c = \mathbb{M}$ and

$$\mathcal{I}_c = \left\{ M \in \mathbb{M} \;\middle|\; \begin{array}{l} \forall (H, S, P, O_1, O_2, O) \in M. \\ (s_1 \rightarrow O_1 = f^*(P)) \wedge (s_2 \rightarrow O_2 = f^*(P)) \end{array} \right\}$$

Mirror or reciprocal. We assume we have an additional operation on hypercontracts, called both mirror and reciprocal, which flips the environments and implementations of a hypercontract: $\mathcal{C}^{-1} = (\mathcal{E}, \mathcal{I})^{-1} = (\mathcal{I}, \mathcal{E})$ and $\mathcal{C}^{-1} = (\mathcal{E}, \mathcal{S})^{-1} = (\mathcal{S}/\mathcal{E}, \mathcal{S})$. This notion gives us, so to say, the hypercontract obeyed by the environment. The introduction of this operation assumes that for every hypercontract $\mathcal{C}$, its reciprocal is also an element of **Contr**. Moreover, we assume that, when the infimum of a collection of hypercontracts exists, the following identity holds:

$$\left(\bigwedge_i \mathcal{C}_i\right)^{-1} = \bigvee_i \mathcal{C}_i{}^{-1}. \tag{4}$$

Hypercontract quotient. Now we consider an operation which answers the following question: given a top-level hypercontract $\mathcal{C}''$ that we want to implement and the hypercontract $\mathcal{C}$ of a partial implementation of the design, what is the largest hypercontract that we can compose with $\mathcal{C}$ in order to meet the top-level specification $\mathcal{C}''$? The *quotient* or residual for hypercontracts $\mathcal{C}$ and $\mathcal{C}''$, written $\mathcal{C}''/\mathcal{C}$, has the universal property $\forall \mathcal{C}'.\ \mathcal{C} \parallel \mathcal{C}' \leq \mathcal{C}''$ if and only if $\mathcal{C}' \leq \mathcal{C}''/\mathcal{C}$. We can obtain a closed-form expression using the reciprocal:

Proposition 1. *The hypercontract quotient obeys* $\mathcal{C}''/\mathcal{C} = \left((\mathcal{C}'')^{-1} \parallel \mathcal{C}\right)^{-1}$.

Example 5 (Running example, quotient). We use the quotient to find the specification of the component that we need to add to the system shown in Figure 1c in order to meet the top level contract $\mathcal{C}$. To compute the quotient, we use (11) from [21]. We let $\mathcal{C}/\mathcal{C}_c = (\mathcal{E}_q,\ \mathcal{I}_q)$ and obtain $\mathcal{E}_q = \mathcal{E} \wedge \mathcal{I}_c$ and

$$\mathcal{I}_q = \left\{ M \in \mathbb{M} \;\middle|\; \begin{array}{l} \exists f \in (2^n \rightarrow 2^o)\ \forall (H, S, P, O_1, O_2, O) \in M. \\ ((s_1 \rightarrow O_1 = f^*(P)) \wedge (s_2 \rightarrow O_2 = f^*(P))) \rightarrow (H = 0 \rightarrow O = f(P)) \end{array} \right\}.$$

We can refine the quotient by lifting any restrictions on the environments, and picking from the implementations the term with $f = f^*$. Observe that f^* is a valid choice for f. This yields the hypercontract $\mathcal{C}_3 = (\mathcal{E}_3, \mathcal{I}_3)$, defined as $\mathcal{E}_3 = \mathbb{M}$ and

$$\mathcal{I}_3 = \left\{ M \in \mathbb{M} \;\middle|\; \begin{array}{l} \forall (H, S, P, O_1, O_2, O) \in M. \\ ((s_1 \rightarrow O_1 = f^*(P)) \wedge (s_2 \rightarrow O_2 = f^*(P))) \rightarrow O = f^*(P) \end{array} \right\}.$$

A further refinement of this hypercontract is $\mathcal{C}_r = (\mathcal{E}_r, \mathcal{I}_r)$, where $\mathcal{E}_r = \mathbb{M}$ and

$$\mathcal{I}_r = \{M \in \mathbb{M} \mid \forall (H, S, P, O_1, O_2, O) \in M.\, ((s_1 \to O{=}O_1) \wedge (s_2 \to O{=}O_2))\}.$$

By the properties of the quotient, composing this hypercontract, which knows nothing about f^*, with $\mathcal{C}_c$ will yield a hypercontract which meets the non-interference hypercontract $\mathcal{C}$. Note that this hypercontract is consistent, i.e., it has implementations (in general, refining may lead to inconsistency).

Merging. Another important operation on hypercontracts is viewpoint merging, or *merging* for short. This operation is used to summarize into a single hypercontract multiple specifications of the same design element (e.g., functionality and performance). Suppose $\mathcal{C}_1 = (\mathcal{E}_1, \mathcal{S}_1)$ and $\mathcal{C}_2 = (\mathcal{E}_2, \mathcal{S}_2)$ are the hypercontracts we wish to merge. Two slightly different operations can be considered as candidates for formalizing viewpoint merging:

– A *weak merge* which is the GLB; and
– A *strong merge* which states that environments of the merger should be environments of both $\mathcal{C}_1$ and $\mathcal{C}_2$ and that the closed systems of the merger are closed systems of both $\mathcal{C}_1$ and $\mathcal{C}_2$. If we let $\mathcal{C}_1 \bullet \mathcal{C}_2 = (\mathcal{E}, \mathcal{I})$, we have

$$\mathcal{E} = \vee\{\mathcal{E}' \in \mathbf{CmpSet} \mid \mathcal{E}' \leq \mathcal{E}_1 \wedge \mathcal{E}_2 \text{ and } \exists \mathcal{C}'' = (\mathcal{E}'', \mathcal{I}'') \in \mathbf{Contr}.\ \mathcal{E}' = \mathcal{E}''\}$$
$$\mathcal{I} = \vee \left\{ \mathcal{I}' \in \mathbf{CmpSet} \;\middle|\; \begin{array}{l} \mathcal{I}' \leq (\mathcal{S}_1 \wedge \mathcal{S}_2)/\mathcal{E} \text{ and} \\ (\mathcal{E}, \mathcal{I}) \in \mathbf{Contr} \end{array} \right\}.$$

The difference is that, whereas the commitment to satisfy $\mathcal{S}_2$ survives under the weak merge when the environment fails to satisfy $\mathcal{E}_1$, no obligation survives under the strong merge. This distinction was proposed in [34] under the name of weak/strong assumptions.

5 Concluding Remarks

We introduced hypercontracts. In hypercontracts, we kept the separation of assumptions and guarantees explicit, as we have found this approach to be the most intuitive for expressing specifications in applications. Hypercontracts extend the scope of AG reasoning to arbitrary, structured hyperproperties, where the structure is dictated by the modeling needs of the adopter of the theory. Imposing no structure allows the development of a theory that supports all hyperproperties. However, limiting expressivity has the advantage of a reduced algorithmic complexity when applying the theory.

Our running example was developed by hand, not aided by a tool. However, having hypercontracts theory was useful in guiding us in this task. For example, atomic properties were easily checked manually with reasonable confidence. However, doing the right chaining was difficult: there, contract theory provided adequate guidelines.

Hypercontracts are a meta-theory: before hypercontracts are used, one must choose a theory of components and a theory of compsets. In this sense, hypercontracts are akin to the meta-theory of Benveniste et al. [5]. This meta-theory builds contracts out of a theory of components, while hypercontracts are built using compsets. This additional algebraic layer allows defining the structured sets of compsets that the theory will use. Moreover, the additional algebraic structure gives hypercontracts the ability to express their formulas using more closed-form expressions than the meta-theory of contracts.

So far, we showed how to instantiate, as hypercontracts, AG contracts and interface automata. Hypercontracts were also used to introduce two new theories of AG reasoning: *interval contracts*, which are similar to AG contracts but come with a "must modality," and more importantly, *conic contracts*, which support arbitrary subset-closed hyperproperties, needed for reasoning about security. The details of these developments are provided in [21,20].

The scope of hypercontracts intersects that of information-flow interfaces [3]. Information-flow interfaces enable assume-guarantee reasoning over information-flow properties. Hypercontracts are complementary to this theory, as they support arbitrary classes of hyperproperties. It is future work to explore the links between these two concepts.

Tom's influence is identifiable in the theory of hypercontracts, and we look forward to continue to get inspiration from his work. Happy festschrift, Tom!

References

1. M. Abadi and L. Lamport. Composing specifications. *ACM Trans. Program. Lang. Syst.*, 15(1):73–132, Jan. 1993.
2. R. Alur, T. A. Henzinger, O. Kupferman, and M. Y. Vardi. Alternating refinement relations. In D. Sangiorgi and R. de Simone, editors, *CONCUR'98 Concurrency Theory*, pages 163–178, Berlin, Heidelberg, 1998. Springer Berlin Heidelberg.
3. E. Bartocci, T. Ferrère, T. A. Henzinger, D. Nickovic, and A. O. Da Costa. Information-flow interfaces. In *International Conference on Fundamental Approaches to Software Engineering*, pages 3–22. Springer, Cham, 2022.
4. A. Benveniste, B. Caillaud, A. Ferrari, L. Mangeruca, R. Passerone, and C. Sofronis. *Multiple Viewpoint Contract-Based Specification and Design*, pages 200–225. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008.
5. A. Benveniste, B. Caillaud, D. Nickovic, R. Passerone, J.-B. Raclet, P. Reinkemeier, A. Sangiovanni-Vincentelli, W. Damm, T. A. Henzinger, and K. G. Larsen. Contracts for system design. *Foundations and Trends® in Electronic Design Automation*, 12(2-3):124–400, 2018.
6. A. Chakrabarti, L. de Alfaro, T. A. Henzinger, and M. Stoelinga. Resource interfaces. In R. Alur and I. Lee, editors, *Embedded Software*, pages 117–133, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg.
7. E. M. Clarke and E. A. Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In *Workshop on Logic of Programs*, pages 52–71. Springer, 1981.
8. M. R. Clarkson and F. B. Schneider. Hyperproperties. *Journal of Computer Security*, 18(6):1157–1210, 2010.

9. W. Damm. Controlling speculative design processes using rich component models. In *Fifth International Conference on Application of Concurrency to System Design (ACSD'05)*, pages 118–119, 2005.
10. A. David, K. G. Larsen, A. Legay, U. Nyman, and A. Wasowski. Timed I/O automata: A complete specification theory for real-time systems. In *Proceedings of the 13th ACM International Conference on Hybrid Systems: Computation and Control*, HSCC '10, page 91–100, New York, NY, USA, 2010. Association for Computing Machinery.
11. L. de Alfaro and T. A. Henzinger. Interface automata. In *Proceedings of the 8th European Software Engineering Conference Held Jointly with 9th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ESEC/FSE-9, page 109–120, New York, NY, USA, 2001. Association for Computing Machinery.
12. L. de Alfaro, T. A. Henzinger, and M. Stoelinga. Timed interfaces. In A. Sangiovanni-Vincentelli and J. Sifakis, editors, *Embedded Software*, pages 108–122, Berlin, Heidelberg, 2002. Springer Berlin Heidelberg.
13. E. W. Dijkstra. Solution of a problem in concurrent programming control. *Commun. ACM*, 8(9):569, sep 1965.
14. L. Doyen, T. A. Henzinger, B. Jobstmann, and T. Petrov. Interface theories with component reuse. In *Proceedings of the 8th ACM International Conference on Embedded Software*, EMSOFT '08, page 79–88, New York, NY, USA, 2008. Association for Computing Machinery.
15. R. W. Floyd. Assigning meanings to programs. *Proceedings of Symposium on Applied Mathematics*, 19:19–32, 1967.
16. J. A. Goguen and J. Meseguer. Security policies and security models. In *1982 IEEE Symposium on Security and Privacy, Oakland, CA, USA, April 26-28, 1982*, pages 11–20, Oakland, CA, USA, 1982. IEEE Computer Society.
17. D. Harel and A. Pnueli. On the development of reactive systems. In K. R. Apt, editor, *Logics and Models of Concurrent Systems*, pages 477–498, Berlin, Heidelberg, 1985. Springer Berlin Heidelberg.
18. T. A. Henzinger, R. Jhala, and R. Majumdar. Permissive interfaces. *SIGSOFT Softw. Eng. Notes*, 30(5):31–40, sep 2005.
19. C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969.
20. I. Incer. *The Algebra of Contracts*. PhD thesis, EECS Department, University of California, Berkeley, May 2022.
21. I. Incer, A. Benveniste, A. Sangiovanni-Vincentelli, and S. A. Seshia. Hypercontracts. *arXiv preprint arXiv:2106.02449*, 2021.
22. I. Incer, A. Benveniste, A. Sangiovanni-Vincentelli, and S. A. Seshia. Hypercontracts. In J. V. Deshmukh, K. Havelund, and I. Perez, editors, *NASA Formal Methods*, pages 674–692, Cham, 2022. Springer International Publishing.
23. L. Lamport. The computer science of concurrency: The early years. *Commun. ACM*, 58(6):71–76, may 2015.
24. K. G. Larsen, U. Nyman, and A. Wąsowski. Modal I/O automata for interface and product line theories. In R. De Nicola, editor, *Programming Languages and Systems*, pages 64–79, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.
25. N. A. Lynch and M. R. Tuttle. An introduction to input/output automata. *CWI Quarterly*, 2:219–246, 1989.
26. I. Mastroeni and M. Pasqua. Verifying bounded subset-closed hyperproperties. In A. Podelski, editor, *Static Analysis*, pages 263–283, Cham, 2018. Springer International Publishing.

27. R. Negulescu. Process spacess. Technical Report CS-95-48, University of Waterloo, 1995.
28. D. L. Parnas. A technique for software module specification with examples. *Commun. ACM*, 15(5):330–336, may 1972.
29. R. Passerone, I. Incer, and A. L. Sangiovanni-Vincentelli. Coherent extension, composition, and merging operators in contract models for system design. *ACM Trans. Embed. Comput. Syst.*, 18(5s), Oct. 2019.
30. A. Pnueli. The temporal logic of programs. In *18th Annual Symposium on Foundations of Computer Science (sfcs 1977)(FOCS)*, pages 46–57, Sept 1977.
31. M. N. Rabe. *A temporal logic approach to information-flow control*. PhD thesis, Universität des Saarlandes, 2016.
32. J. Raclet. Residual for component specifications. *Electr. Notes Theor. Comput. Sci.*, 215:93–110, 2008.
33. J.-B. Raclet, E. Badouel, A. Benveniste, B. Caillaud, A. Legay, and R. Passerone. A modal interface theory for component-based design. *Fundamenta Informaticae*, 108(1-2):119–149, 2011.
34. A. Sangiovanni-Vincentelli, W. Damm, and R. Passerone. Taming Dr. Frankenstein: Contract-based design for cyber-physical systems. *European Journal of Control*, 18(3):217–238, 2012.
35. J. Sifakis. Rigorous system design. *Foundations and Trends® in Electronic Design Automation*, 6(4):293–362, 2013.
36. A. M. Turing. On checking a large routine. In *Report of a Conference on High-Speed Automatic Calculating Machines*, pages 67–69, Cambridge, 1949. University Mathematical Laboratory.