









# Information-flow Interfaces<sup>\*</sup>

Ezio Bartocci<sup>1</sup> , Thomas Ferrère<sup>2</sup> , Thomas A. Henzinger<sup>3</sup> ,  
Dejan Nickovic<sup>4</sup> , and Ana Oliveira da Costa<sup>1</sup>  

<sup>1</sup> Technische Universität Wien, Vienna, Austria  
{ezio.bartocci, ana.costa}@tuwien.ac.at

<sup>2</sup> Imagination Technologies, Kings Langley, UK  
thomas.ferrere@imgtec.com

<sup>3</sup> IST Austria, Klosterneuburg, Austria  
tah@ist.ac.at

<sup>4</sup> AIT Austrian Institute of Technology, Vienna, Austria  
dejan.nickovic@ait.ac.at

**Abstract.** Contract-based design is a promising methodology for taming the complexity of developing sophisticated systems. A formal contract distinguishes between *assumptions*, which are constraints that the designer of a component puts on the environments in which the component can be used safely, and *guarantees*, which are promises that the designer asks from the team that implements the component. A theory of formal contracts can be formalized as an *interface theory*, which supports the composition and refinement of both assumptions and guarantees. Although there is a rich landscape of contract-based design methods that address functional and extra-functional properties, we present the first interface theory that is designed for ensuring system-wide security properties. Our framework provides a refinement relation and a composition operation that support both incremental design and independent implementability. We develop our theory for both *stateless* and *stateful* interfaces. We illustrate the applicability of our framework with an example inspired from the automotive domain.

**Keywords:** Contract-based design, Interface Theory, Hyperproperties, Information-flow.

## 1 Introduction

The rise of pervasive information and communication technologies seen in cyber-physical systems, internet of things, and blockchain services has been accompanied by a tremendous growth in the size and complexity of systems [28]. Subtle dependencies involving multiple architectural layers and unforeseen environmental interactions can expose these systems to cyber-attacks. This problem is further exacerbated by the heterogeneous nature of their constituent components,

---

<sup>\*</sup> This project has received funding from the European Unions Horizon 2020 research and innovation programme under grant agreement No 956123 and was funded in part by the FWF project W1255-N23 and by the ERC-2020-AdG 101020093.

which are often developed independently by different teams or providers. In such a scenario, defining and enforcing security requirements across components at an early stage of the design process becomes a necessity. This engineering approach is called *security-by-design*. Although in recent years there has been impressive progress in the verification of security properties for individual system components, the science of compositional security design [22,23] is still in its infancy.

Security policies are usually enforced by restricting the flow of information in a system [30]. *Information-flow policies* define which information a user or a software/hardware component is allowed to observe or to interfere with while interacting with another component.

The goal of information-flow control is to ensure that a system as a whole satisfies the desired policies. It is especially challenging to verify that there are no side-channels or implicit flows that violate a given policy. For example, in a modern car, the tight coupling between the cyber and the physical components allows an attacker to infer computational properties, such as secrets used for encryption, from side-channels, such as power consumption and electromagnetic radiation [17]. Moreover, the increasing connectivity of automotive systems with their environment makes it easier for the attacker to gather data about the system behavior. The attacker can use this data to exploit weaknesses of the system implementation and gain control over the system [32,7]. These attacks often rely on analyzing and comparing multiple observations to deduce protected information. From a formal-language perspective, such security vulnerabilities are not characterized by properties of a single system execution, but rather by properties of sets of execution traces, which are called *hyperproperties* [12].

The rigorous design of systems that satisfy information flow requirements is essential from the security perspective. This activity can be supported by the verification of information flow properties, a well-studied problem with a rich landscape of both theory and tools, ranging from language-based [29,18,15,11] to simulation-based [25] approaches. Nevertheless, the existing verification solutions do not address two important aspects. First, components in complex systems are often heterogeneous and cannot be analysed with a single verification tool. Moreover, it is not clear how to combine component verification outcomes to infer system-level information-flow properties. Second, existing methods do not provide guidelines on which information flow properties need to be verified against individual components to provide system-level guarantees regarding leakage of information.

In this paper, we present a *contract-based design* [8] approach for the structural aspect of information-flow policies. Contract-based design provides a formal framework for building complex systems from individual components, mixing both top-down and bottom-up steps. A top-down step decomposes and refines system-wide requirements; a bottom-up step assembles a system by combining available components. A formal contract distinguishes between *assumptions*, which are constraints that the designer of a component puts on the environments in which the component can be used safely, and *guarantees*, which are promises that the designer asks from the team that implements the component. A theory

of formal contracts can be formalized as an *interface theory*, which supports the composition and refinement of both assumptions and guarantees [2,3,31]. While there is a rich landscape of interface theories for functional and extra-functional properties [10,4,13,20], we present the first interface theory that is designed for ensuring system-wide security properties, thus paving the way for a science of safety and security co-engineering.

The focus on the structural aspects of information flow and abstraction from concrete semantics enables compositional reasoning in presence of heterogeneous components and is complementary to the existing body of work on information flow verification. A different component implementation verified under different semantics could result in different flows being detected. However, after deriving the component flows from the implementation under some concrete semantics, the theory can be agnostic about the underlying semantic interpretation. Hence it enables the design of secure systems from trusted components by abstracting away how information flows and by focusing on whether it can flow at all. In essence, our approach enables to decompose system-level information flow requirements and derive component properties that need to hold, thus providing a divide-and-conquer procedure for organizing verification tasks.

Our theory is based on information-flow assumptions as well as information-flow guarantees. As an interface theory, our theory supports both *incremental design* and *independent implementability* [3]. Incremental design allows the composition of different system parts, each coming with their own assumptions and guarantees, without requiring additional knowledge of the overall design context. Independent implementability enables the separate refinement of different system parts by different teams that, without gaining additional information about each other’s design choices, can still be certain that their designs, once combined, preserve the specified system-wide requirements. While in previous interface theories, the environment of a component is held responsible for meeting assumptions, and the implementation of the component for the guarantees, there are cases of information-flow violations for which blame cannot be assigned uniquely to the implementation or the environment. In information-flow interfaces we therefore introduce, besides assumptions and guarantees, a new, third type of constraint—called *properties*—whose enforcement is the shared responsibility of the implementation and the environment.

We develop our framework for both *stateless* and *stateful* interfaces. Stateless information-flow interfaces are built from primitive information-flow constraints—assumptions, guarantees, and properties—of the form “the value of a variable  $x$  is always independent of the value of another variable  $y$ .” Stateful information-flow interfaces add a temporal dimension, e.g., “the value of  $y$  is independent of  $x$  until the value of  $z$  is independent of  $x$ .” The temporal dimension is introduced through a natural notion of state and state transition for interfaces, not through logical operators. We prove that our calculus of information-flow interfaces satisfies the principles of incremental design and independent implementability.

## 2 Application Example

We showcase the applicability of our theory with an example from the automotive industry: a stepwise design of a shared communication infrastructure (a bus) from distance warners and a wheel sensor to the braking system and the odometer. We adapted this use-case from the industrial case study presented by Marcus Mikulcak et al. [25]. The main goal of this system design is to ensure the integrity of a communication channel used to perform a safety-critical functionality. We consider two integrity levels, high and low, to characterize functionalities in our system. Then, we want to guarantee that data exchanged by high-integrity functionalities is not compromised by low-integrity functions.

Distance warners sense the car’s proximity to other objects and send their analysis to other components. In our example, we have two distance warners, at the front and the back of the car, that use the shared bus to communicate with the braking system. The wheel sensor senses the wheel rotations and sends this information through the shared bus to the odometer. The braking system is a high-integrity system since it performs safety-critical functions. Hence the communication channel between the distance warners and the braking system is classified with high-integrity, while the communication between the wheel sensor and the odometer is low-integrity. Thus, data sent by the wheel sensor should not interfere with the high-integrity channel to prevent distance warnings sent to the braking system from being delayed or lost. The main goal of our design process is to guarantee that the closed system requirement that *information from the wheel sensor does not flow to the braking system* is propagated accordingly to subsystems through successive decomposing and refinement steps.

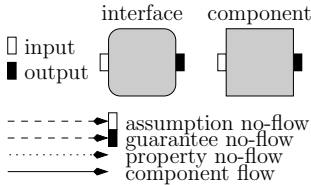


Fig. 1: Representation of the objects in our theory.

Figure 1 shows the graphical representation we adopt throughout the paper for the objects in our theory. We represent the open system no-flows requirements with dashed arrows. Then, arrows to input ports are *assumptions* while arrows to output parts are *guarantees*. The closed system no-flows, *properties*, are represented as dotted arrows. To improve the readability of the drawings, it is implicit that for each drawn property, we have the same guarantee over the open system. When it is clear from the context we may omit port(s) names.

We present, in Fig. 2, the stepwise design of the security requirement that data from the wheel sensor, *wheel\_tick*, should not flow to the target of the distance warners, *distw\_f\_t* and *distw\_b\_t*. The first interface in Fig. 2 includes two properties that specify this security requirement. The system is then decomposed into the *sending subsystems* (warners and wheel sensor), the *shared bus*, and the *receiving subsystems*.

Naturally, we keep the two properties from the first interface as properties in the Bus interface. However, this natural decomposition *does not define a well-formed interface* according to our theory because the properties in the Bus interface cannot be satisfied given the interface’s current assumption and

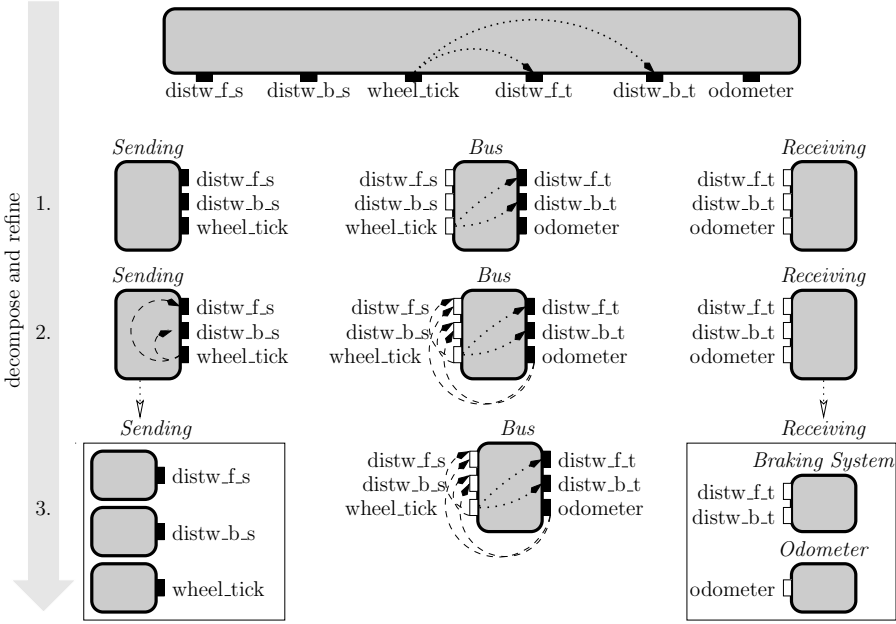


Fig. 2: Top-down design of a shared communication infrastructure used by two distance warners,  $distw\_f\_s$  and  $distw\_b\_s$ , and a wheel sensor,  $wheel\_tick$ , to communicate with the braking system,  $distw\_f\_t$  and  $distw\_b\_t$ , and the odometer,  $odometer$ , respectively.

guarantee. As the environment allows a flow from  $wheel\_tick$  to the source of the front distance warner,  $wheel\_tick \rightsquigarrow distw\_f\_s$  then, with the flow allowed by the guarantee from a distance warner source to its target, we have the flow  $wheel\_tick \rightsquigarrow distw\_f\_s \rightsquigarrow distw\_f\_t$ . This flow is forbidden by the interface's properties. If we specified the no-flow properties in the Bus interface as guarantees, then the interface would be well-formed. However, the composition of the three subsystems would not satisfy the initial specification because guarantees only apply to implementations of their interface, and the flow described above would still be allowed in the composition of the three subsystems. This illustrates two applications of the information-flow interface theory: *to detect inconsistent no-flow specifications* and *faulty decompositions*. Moreover, when an interface is not well-formed we can provide a witness for the property violation. We can use this witness to *guide the refinement of an ill-formed interface* into a well-formed one.

In the second step of our refinement, in Figure 2, we add the missing assumptions to the Bus interface. Our notion of *composition compatibility between interfaces* requires that the Sending interface includes guarantees that implies the Bus assumptions, as the Sending interface will be part of the Bus environment. At this point, with a certified decomposition of the original specification, our theory guarantees that each subsystem can now be further *refined independently* (possibly by different teams). The last step illustrates an independent refinement of the Sending and the Receiving interfaces.

In Fig. 3, we present the stateful view of the system, which requires that the system satisfies the composition of the Sending, the Bus, and the Receiving interfaces derived in Fig. 2 at all times. We present, as well, a refinement of that specification, which requires that in each time point only one of the sending components can use the bus. The interfaces that define each state are named after the sending component that can use the bus (e.g. in the state  $S_{\text{wheel}}$  only the *wheel.tick* can use the bus). If the access to the bus is mutually exclusive, then we can simplify the assumptions on the environment in the Bus interface. With more guarantees on the implementations we need fewer assumptions to satisfy the properties.

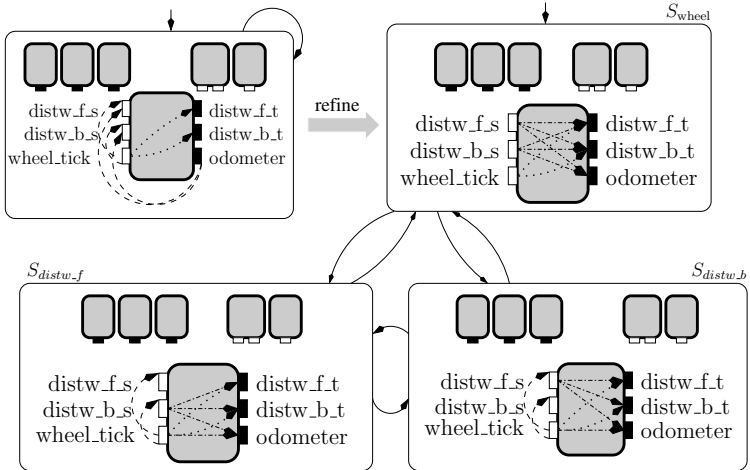


Fig. 3: Design of mutually exclusive shared communication infrastructure for distance warners and the wheel odometer. Each state is defined by the composition of the interfaces inside.

Finally, the components of our system can be, for instance, the Simulink and Stateflow models provided to the authors [25] by their industrial partners. We can then use the tool introduced in their work to verify whether these components implement the stateful interfaces we derived.

In summary, our framework defines relations on both stateless and stateful interfaces specifying information-flow policies that allow to check if: (i) a given interface refines (or abstracts) the current specification; (ii) two interfaces are compatible for composition; (iii) a specification is consistent; (iv) information-flows in a component define an implementation of a given interface; and (v) a system decomposition refines the system specification.

### 3 Stateless Information-flow Interfaces

In this section, we introduce a stateless interface and component algebra for secure information flow. Information flows between two variables when the value of one influences the other.

We are interested in the *structural* properties of information flow within a system and define relations abstracting flows, *flow relations*, as being both reflexive and transitively closed. An *information-flow component* abstracts the implementation of a system by a *flow relation*. An *information-flow interface* specifies forbidden flows in an open system by defining three kinds of constraints: assumptions, guarantees, and properties. The *assumption* characterizes flows that we assume are not part of the environment while the *guarantee* describes all flows the system forbids and that are local to it. The *property* qualifies the forbidden flows at the interaction between the system and its environment. Hence, it represents a requirement on the closed system that needs to be enforced by guarantees on the open system and assumptions on its environment.

**Definition 1.** Let  $X$  and  $Y$  be disjoint sets of input and output variables, respectively, with  $Z = X \cup Y$  the set of all variables. A stateless information-flow component is a tuple  $(X, Y, \mathcal{M})$ , where  $\mathcal{M} \subseteq Z \times Y$  is a (reflexive and transitive) flow relation, called flows. A stateless information-flow interface is a tuple  $(X, Y, \mathcal{A}, \mathcal{G}, \mathcal{P})$ , where:  $\mathcal{A} \subseteq Z \times X$  is a relation, called assumption;  $\mathcal{G} \subseteq Z \times Y$  is a relation, called guarantee; and  $\mathcal{P} \subseteq Z \times Y$  is a relation, called property.

Given an interface  $F$  we are interested in components that do not implement flows forbidden by either the interface guarantees (called *implementations of  $F$* ) or the interface assumptions (called *environments of  $F$* ).

**Definition 2.** A component  $f_{\mathcal{E}} = (Y, X, \mathcal{E})$  is called an environment of  $F = (X, Y, \mathcal{A}, \mathcal{G}, \mathcal{P})$ . An environment is admissible for  $F$ , denoted by  $f_{\mathcal{E}} \models F$ , iff  $\mathcal{E} \subseteq \bar{\mathcal{A}}$ . A component  $f = (X, Y, \mathcal{M})$  implements the interface  $F$ , denoted by  $f \models F$ , iff  $\mathcal{M} \subseteq \bar{\mathcal{G}}$ .

*Example 1.*

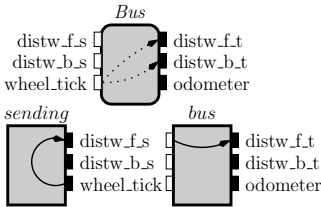


Fig. 4: Interface *Bus* with an implementation, *bus*, and an admissible environment, *sending*.

In Figure 4, we have the first refinement of the interface *Bus* from our application example. The *Bus* interface specifies the requirement on the closed system (using properties) that there are no-flows from *wheel\_tick* to both *distw\_f\_s* and *distw\_b\_s*. The *Bus* interface specifies this requirement as a guarantee on the open system, too. Then, the *bus* component is an implementation of *Bus* because it has only a flow from *distw\_f\_s* to *distw\_f\_t*, which is not in the guarantees of the *Bus* interface. *Bus* does not have any assumptions, then the *sending* component is an environment for *Bus*.

When we compose the components *sending* and *bus*, there is a flow from *wheel\_tick* to *distw\_f\_t*, which is in the properties of the *Bus*. Hence the assumption and guarantee specified over the open system are not enough to ensure the property over the closed system. The composition of these two components witness that the *Bus* interface is not well-formed.

An information-flow interface is *well-formed* when it has at least one implementation and one admissible environment. Therefore, all of its relations must be irreflexive. We refer to irreflexive relations as *no-flow* relations. A well-formed interface ensures, additionally, that an interface property is *consistent* with its assumptions and guarantees. An interface property is not consistent when the flow relation defined by the composition of one of the interface's admissible environments with one of its implementations includes a pair specified in the interface property. To check whether the property is consistent, we compute the *flow relation of the closed system defined by an interface  $F$* , which includes all flows that are in the composition of any of the interface's admissible environments with one of its implementations. The main challenge is that, in general, the complement of an interface's guarantee (assumption) may not define the flow relation of any of its implementations (environments). Hence there may be no maximal implementation or admissible environment for a given interface.

*Example 2.*

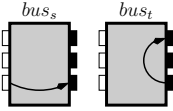


Fig. 5:  $Bus$  implementations.

In Figure 5, we have two components,  $bus_s$  and  $bus_t$ , that implement the interface  $Bus$  from the previous example. A maximal implementation of  $Bus$  must include the flows in both  $bus_s$  and  $bus_t$ . As flows are transitively closed, the maximal implementation would include a flow from  $wheel\_tick$  to  $distw\_f\_t$ , which violates the  $Bus$  guarantees.

Given that we do not have maximal implementations and maximal admissible environments, then we cannot characterize all flows of the closed system defined by an interface  $F$  by computing the transitive closure of all pairs in the complement of  $F$ 's assumption and guarantee  $-(\overline{\mathcal{A}} \cup \overline{\mathcal{G}})^*$ . This approach would yield more flows than the flows of the closed system defined by  $F$ . Instead, we consider all pairs  $(z, z')$  such that there exists a path from  $z$  to  $z'$  that alternates between flows in the complement of the assumption,  $\overline{\mathcal{A}}$ , and the guarantee,  $\overline{\mathcal{G}}$ . We define this notion below as the *composition between no-flow relations*. In Proposition 1 we prove that this definition captures our intended relation between an interface property and its environments and implementations.

**Definition 3.** A no-flow relation  $\mathcal{N} \subseteq (A \cup B) \times B$  is an irreflexive relation, and its complement is  $\overline{\mathcal{N}} = ((A \cup B) \times B) \setminus \mathcal{N}$ . Let  $\mathcal{N} \subseteq (A \cup B) \times B$  and  $\mathcal{N}' \subseteq (A' \cup B') \times B'$  be two no-flow relations. The set of flows defined by their composition is  $\mathcal{N} \bullet \mathcal{N}' = (Id_{A' \cup B'} \cup \overline{\mathcal{N}}) \circ (\overline{\mathcal{N}'} \circ \overline{\mathcal{N}})^* \circ (Id_B \cup \overline{\mathcal{N}'})$ , where  $Id_Z = \{(z, z) \mid z \in Z\}$  and  $R \circ R' = \{(z, z'') \mid (z, z') \in R \text{ and } (z', z'') \in R'\}$  is the usual composition between relations.

We have now all the ingredients to define well-formed interfaces.

**Definition 4.** An interface  $(X, Y, \mathcal{A}, \mathcal{G}, \mathcal{P})$  is well-formed iff  $\mathcal{A}$ ,  $\mathcal{G}$  and  $\mathcal{P}$  are no-flow relations; and the property is consistent, i.e.  $(\mathcal{A} \bullet \mathcal{G}) \cap \mathcal{P} = \emptyset$ .

**Proposition 1.** For all well-formed interfaces  $F = (X, Y, \mathcal{A}, \mathcal{G}, \mathcal{P})$ , and for all components  $f = (X, Y, \mathcal{M})$  and  $f_{\mathcal{E}} = (Y, X, \mathcal{E})$ : if  $f$  implements  $F$ ,  $f \models F$ , and



$f_{\mathcal{E}}$  is an admissible environment of  $F$ ,  $f_{\mathcal{E}} \models F$ , then their combined flows are consistent with the property of  $F$ ,  $(\mathcal{M} \cup \mathcal{E})^* \cap \mathcal{P} = \emptyset$ .

### 3.1 Composition and Incremental Design

We now present how to *compose* components and interfaces. We introduce a *compatibility* predicate that checks whether the composition of two interfaces is a well-formed interface. We prove that these two notions support the *incremental design* of systems.

The different types of variables between interfaces  $F$  and  $F'$  are defined as  $Y_{F,F'} = Y \cup Y'$ ,  $X_{F,F'} = (X \cup X') \setminus Y_{F,F'}$ , and  $Z_{F,F'} = Y_{F,F'} \cup X_{F,F'}$ . The same definition applies to components  $f$  and  $f'$ . The *composition of components*  $f$  and  $f'$  is the reflexive and transitive closure of the union of the individual component flows, i.e.  $f \otimes f' = (X_{f,f'}, Y_{f,f'}, (\mathcal{M} \cup \mathcal{M}')^*)$ . We present interface composition by defining separately  $\mathcal{A}$ ,  $\mathcal{G}$  and  $\mathcal{P}$  of the composed interface.

We compose interfaces through their shared variables. *Shared variables between two interfaces* are all variables that are an input variable in one of the interfaces and an output variable in the other one. The *composite flows* between two interfaces is the set with all flows that are in the composition of any of their implementations. As for the definition of flows in the closed system defined by an interface, the composite flows are the composition of the guarantees of the interfaces being composed (as defined in Definition 3). The composition of two interfaces should not restrict their sets of implementations, thus the *composite guarantees* are the complement of the composite flows.

**Definition 5.** Let  $F = (X, Y, \mathcal{A}, \mathcal{G}, \mathcal{P})$  and  $F' = (X', Y', \mathcal{A}', \mathcal{G}', \mathcal{P}')$  be two interfaces. Their composite flows are  $\overline{\mathcal{G}}_{F,F'} = \mathcal{G} \bullet \mathcal{G}'$ . The composite guarantees of  $F$  and  $F'$  are defined as  $\mathcal{G}_{F,F'} = (Z_{F,F'} \times Y_{F,F'}) \setminus \overline{\mathcal{G}}_{F,F'}$ , also denoted by  $\mathcal{G}_{F \otimes F'}$ .

The assumption of an interface resulting from the composition of multiple interfaces is the weakest condition on the environment that allows the interfaces being composed to work together. Additionally, it must support incremental design, i.e. the admissibility of an environment must be independent of the order in which the interfaces are composed.

Naturally, all assumptions of each interface must be considered during composition. However, not all of them can be kept as assumptions of the composite interface, because shared variables will be output variables of the composition. If the environment can still influence the information flow to a shared variable, then we may need to add assumptions to prevent such a flow. *Propagated assumptions* between two interfaces are derived by looking in their respective assumptions for no-flow pairs pointing to a shared variable.

*Example 3.*

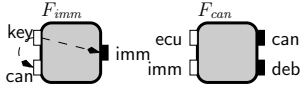


Fig. 6: Propagating assumptions.

In Figure 6, we depict an interface specifying information-flow policies for a car immobilizer,  $F_{imm}$ , along with an interface for a Controller Area Network (CAN bus),  $F_{can}$ . Interface  $F_{imm}$  has only one assumption that key does not flow to can. In this design, the immobilizer uses the CAN to communicate with the car electronic control unit (ECU). Our goal is to compose both interfaces.

These interfaces share the port can. Thus, can will be an output port of their composition. The interface  $F_{can}$  cannot guarantee that the only assumption in  $F_{imm}$  is satisfied after composition because it does not have a port key. As we are working with open systems and assume that the environment is helpful, we can add further assumptions to ensure the correctness of this composition. For example, we can add assumptions that prevent key from flowing to an input port in  $F_{can}$  that can flow to can. Such flows could be part of a flow from key to can, which would violate the assumption we want to enforce. In this case, we note that in  $F_{can}$  information in ecu can flow to can. So, the composite interface needs to include the assumption that key does not flow to ecu. This is a *propagated assumption*.

**Definition 6.** *The set of assumptions propagated from  $F = (X, Y, \mathcal{A}, \mathcal{G}, \mathcal{P})$  to  $F' = (X', Y', \mathcal{A}', \mathcal{G}', \mathcal{P}')$  is  $\hat{\mathcal{A}}_{F \rightarrow F'} = \{(z, z') \mid \exists s \in X \cap Y' \text{ s.t. } (z, s) \in \mathcal{A} \text{ and } (z', s) \in \bar{\mathcal{G}}_{F, F'}\}$ . The set with all propagated assumptions of  $F$  and  $F'$  is  $\hat{\mathcal{A}}_{F, F'} = \hat{\mathcal{A}}_{F \rightarrow F'} \cup \hat{\mathcal{A}}_{F' \rightarrow F}$ . The composite assumptions of  $F$  and  $F'$  are defined as  $\mathcal{A}_{F, F'} = (\mathcal{A} \cup \mathcal{A}' \cup \hat{\mathcal{A}}_{F, F'}) \cap (Z_{F, F'} \times X_{F, F'})$ , also denoted by  $\mathcal{A}_{F \otimes F'}$ .*

*Example 4.* From the example before, information from the ports ecu, imm and deb can all flow to can. So, they are flows in the composite interface and, by Definition 5,  $\{(\text{ecu}, \text{can}), (\text{imm}, \text{can}), (\text{deb}, \text{can})\} \subseteq \bar{\mathcal{G}}_{F_{imm}, F_{can}}$ . Then,  $\hat{\mathcal{A}}_{F_{imm} \rightarrow F_{can}} = \{(\text{key}, \text{ecu}), (\text{key}, \text{imm}), (\text{key}, \text{deb})\}$ . From those assumptions only (key, ecu) points to a variable in  $X_{F, F'}$ , so  $\mathcal{A}_{F_{imm}, F_{can}} = \{(\text{key}, \text{ecu})\}$ .

The properties of the composition contains all properties of each interface being composed. They include, additionally, all derived properties from the assumptions and guarantees of the composite. *Derived properties* are guarantees that hold under any admissible environment. They are defined by all pairs  $(z, y)$  in an interface guarantee s.t. there is no combination of flows allowed by its assumptions and guarantees that creates a flow from  $z$  to  $y$ . Then, the *derived properties* of an assumption  $\mathcal{A}$  and guarantee  $\mathcal{G}$  is defined as  $\mathcal{P}^{\mathcal{A}, \mathcal{G}} = \mathcal{G} \setminus (\mathcal{A} \bullet \mathcal{G})$ . The *composite properties* of  $F$  and  $F'$  are  $\mathcal{P}_{F, F'} = \mathcal{P} \cup \mathcal{P}' \cup \mathcal{P}^{\mathcal{A}_{F, F'}, \mathcal{G}_{F, F'}}$ .

**Definition 7.** *The composition of two interfaces  $F$  and  $F'$  is the interface:  $F \otimes F' = (X_{F, F'}, Y_{F, F'}, \mathcal{A}_{F, F'}, \mathcal{G}_{F, F'}, \mathcal{P}_{F, F'})$ , where  $\mathcal{A}_{F, F'}$  is defined in Definition 6,  $\mathcal{G}_{F, F'}$  defined in Definition 5 and  $\mathcal{P}_{F, F'}$  in the previous paragraph.*

We allow composition for any two arbitrary interfaces. However, not all compositions result in a well-formed interface. We define next the notions of two

interfaces being *composable* and *compatible*. Composability imposes the syntactic restriction that both interface's output variables are disjoint. Compatibility captures the semantic requirement that whenever an interface  $F$  provides inputs to another interface  $F'$ , then  $F'$  needs to include guarantees that imply the assumptions of  $F$ .

**Definition 8.** *Two interfaces  $F = (X, Y, \mathcal{A}, \mathcal{G}, \mathcal{P})$  and  $F' = (X', Y', \mathcal{A}', \mathcal{G}', \mathcal{P}')$  are composable iff  $Y \cap Y' = \emptyset$ . The interfaces  $F$  and  $F'$  are compatible, denoted  $F \sim F'$  iff they are composable and  $((\mathcal{A} \cup \mathcal{A}') \cap (Z_{F, F'} \times Y_{F, F'})) \subseteq \mathcal{G}_{F, F'}$ .*

Clearly, both the composition operator and the compatibility relation are commutative. Additionally, we prove that composition preserves well-formedness and that it supports incremental design of systems. The full proofs are in the appendix.

**Theorem 1.** *Let  $F$  and  $F'$  be well-formed interfaces. If the interfaces are compatible,  $F \sim F'$ , then their composition,  $F \otimes F'$ , defines a well-formed interface.*

**Theorem 2 (Incremental design).** *Let  $F$ ,  $F'$  and  $F''$  be interfaces. If  $F \sim F'$  and  $(F \otimes F') \sim F''$ , then  $F' \sim F''$  and  $F \sim (F' \otimes F'')$ .*

*Proof.* We proved first that composite assumptions are associative. We assume that  $F \sim F'$  and  $(F \otimes F') \sim F''$ . The most interesting case is when  $(z, s)$  is an assumption of  $F$  and  $s$  is a shared variable between  $F$  and  $F' \otimes F''$ . Then, we need to prove that  $(z, s) \in \mathcal{G}_{F, F' \otimes F''}$ . We prove this by assuming towards a contradiction that  $(z, s) \in \overline{\mathcal{G}}_{F, F' \otimes F''}$ . We illustrate it in Figure 7.

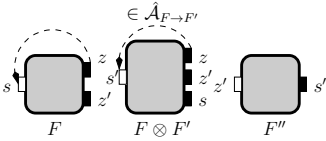


Fig. 7: Incremental design.

By composite flows being associative,  $(z, s) \in \overline{\mathcal{G}}_{F \otimes F', F''}$ . By  $(z, s)$  being an assumption of  $F$  and  $(s', s) \in \overline{\mathcal{G}}_{F \otimes F'}$ , then we have the derived assumption  $(z, s') \in \hat{\mathcal{A}}_{F \rightarrow F'}$  and, so  $(z, s') \in \mathcal{A}_{F \otimes F'}$ . Moreover,  $(z, s') \in \overline{\mathcal{G}}_{F \otimes F', F''}$ , because  $z$  can flow to  $s'$  when  $F \otimes F'$  is composed with  $F''$ . This contradicts our initial assumption that  $(F \otimes F') \sim F''$ .  $\square$

We prove additionally that composition is associative for compatible interfaces.

**Theorem 3.** *If  $F \sim F'$  and  $F \otimes F' \sim F''$ , then  $(F \otimes F') \otimes F'' = F \otimes (F' \otimes F'')$ .*

Finally, we show that flows resulting from the composition of any components that implement two given interfaces are allowed by the composition of these interfaces.

**Proposition 2.** *For all two interfaces  $F$  and  $F'$ , and all two components  $f = (X, Y, \mathcal{M})$  and  $f' = (X', Y', \mathcal{M}')$  that implement them,  $f \models F$  and  $f' \models F'$ , then the composition of the components implements the composition of the interfaces,  $f \otimes f' \models F \otimes F'$ .*

### 3.2 Refinement and Independent Implementability

We now define a refinement relation between interfaces. Intuitively, an interface  $F'$  refines  $F$  iff  $F'$  admits more environments than  $F$ , while possibly constraining its implementations.

**Definition 9.** *Interface  $F' = (X', Y', \mathcal{A}', \mathcal{G}', \mathcal{P}')$  refines  $F = (X, Y, \mathcal{A}, \mathcal{G}, \mathcal{P})$ , written  $F' \preceq F$ , when  $\mathcal{A}' \subseteq \mathcal{A}$ ,  $\mathcal{G} \subseteq \mathcal{G}'$  and  $\mathcal{P} \subseteq \mathcal{P}'$ .*

Let  $F$  and  $F'$  be interfaces s.t.  $F' \preceq F$ . Let  $f = (X, Y, \mathcal{M})$  and  $f_{\mathcal{E}} = (Y, X, \mathcal{E})$  be components. Then, (a) If  $f \models F'$ , then  $f \models F$ ; and (b) if  $f_{\mathcal{E}} \models F$ , then  $f_{\mathcal{E}} \models F'$ .

Additionally, we show below that refinement and composition supports independent implementability.

**Theorem 4 (Independent implementability).** *For all well-formed interfaces  $F'_1, F_1$  and  $F_2$ , if  $F'_1 \preceq F_1$  and  $F_1 \sim F_2$ , then  $F'_1 \sim F_2$  and  $F'_1 \otimes F_2 \preceq F_1 \otimes F_2$ .*

*Proof.* The challenging part is to prove that the refined composite contains all properties of the abstracted one, i.e.  $\mathcal{P}_{F_1 \otimes F_2} \subseteq \mathcal{P}_{F'_1 \otimes F_2}$ . We prove by induction on  $n \in \mathbb{N}$  that if a pair of variables  $(z, y)$  cannot be defined by assume-guarantee paths of size at most  $n$  of the abstract composition, then it cannot be defined by assume-guarantee paths of size at most  $n$  of the refined composition. We can see easily for the base case. If for all  $(z, s) \in \overline{\mathcal{A}}_{F_1, F_2}$  s.t. there exists  $(s, y) \in \mathcal{G}_{F_1, F_2}$ , then, by  $F'_1 \preceq F_1$ , it follows that for all  $(z, s) \in \overline{\mathcal{A}}_{F'_1, F_2}$  there exists  $(s, y) \in \mathcal{G}_{F'_1, F_2}$ . Hence if  $(z, y) \notin \overline{\mathcal{A}}_{F_1, F_2} \circ \overline{\mathcal{G}}_{F_1, F_2}$ , then  $(z, y) \notin \overline{\mathcal{A}}_{F'_1, F_2} \circ \overline{\mathcal{G}}_{F'_1, F_2}$  as well.  $\square$

### 3.3 Discussion

*Properties.* In this work, we consider transitively closed flows. In this setting, in an open system, information can flow from  $z$  to  $z'$  by flowing from  $z$  to  $s$  through the environment, and then from  $s$  to  $z'$  through one of its implementations. As our algebra focuses on the design of structural requirements of no-flows in open systems, it needs to support the specification of global no-flow requirements. We made them explicit by introducing properties. If we did not include properties in our interfaces, then either assumptions or guarantees would need to take over the role of specifying global no-flows. Let's assume that, alternatively, guarantees would be interpreted as global no-flows. Then, to support incremental design, the compatibility criteria between interfaces would turn out to be overly restrictive, with intuitive and correct designs being considered incompatible. This led us to the distinction between guarantees and properties, where properties may be supported by assumptions on the environment that can restrict the set of compatible interfaces. In other words, the main advantage of having properties is that the designer can choose how to split the responsibilities between the environment and the implementations to satisfy a global no-flow.

*Semantics.* The structural approach that abstracts away semantic considerations is an important feature of our theory. The practicability of our approach lies in the support for the design of such requirements by decoupling the design process from (its orthogonal) semantic considerations. Hence, our approach does not deny semantics, but rather separates the design of specifications from component implementation concerns. The presented approach even allows using tailored semantics and tools for different parts of the design. For example, at the bottom (component) level, no-flows and flows relations can be instantiated with different semantic interpretations. After deriving the component no-flows from the implementation under a concrete semantics, the theory can be agnostic about the underlying semantic interpretation and can focus on whether there exists a flow at all.

## 4 Stateful Information-Flow Interfaces

We extend our theory with stateful components and interfaces. These are transition systems in which each state is a stateless component or interface, respectively.

**Definition 10.** *Let  $X$  and  $Y$  be disjoint sets of input and output variables, respectively, with  $Z = X \cup Y$  the set of all variables. Let  $Q$  be a set of states with  $\hat{q} \in Q$  being the initial state and  $\delta : Q \rightarrow 2^Q$  be a transition relation. A stateful information-flow component  $\mathbf{f}$  is a tuple  $(X, Y, Q, \hat{q}, \delta, \mathbb{M})$ , where  $\mathbb{M} : Q \rightarrow 2^{Z \times Y}$  is a state labeling such that for all states  $q \in Q$ ,  $\mathbb{M}(q)$  defines a flow relation. We denote by  $\mathbf{f}(q) = (X, Y, \mathbb{M}(q))$  the stateless component implied by the labeling of  $q$ . A stateful information-flow interface  $\mathbb{F}$  is a tuple  $(X, Y, Q, \hat{q}, \delta, \mathbb{A}, \mathbb{G}, \mathbb{P})$ , where  $\mathbb{A} : Q \rightarrow 2^{Z \times X}$  is called assumption;  $\mathbb{G} : Q \rightarrow 2^{Z \times Y}$  is called guarantee; and  $\mathbb{P} : Q \rightarrow 2^{Z \times Y}$  is called property. For each state  $q \in Q$  we denote by  $\mathbb{F}(q) = (X, Y, \mathbb{A}(q), \mathbb{G}(q), \mathbb{P}(q))$  the stateless interface implied by the assumption, guarantee and property of  $q$ .*

A stateful interface  $\mathbb{F}$  is well-formed iff  $\mathbb{F}(\hat{q})$  is a well-formed stateless interface, and for all  $q \in Q$  reachable from  $\hat{q}$  the stateless interface  $\mathbb{F}(q)$  is well-formed. In what follows,  $\mathbb{F} = (X, Y, Q, \hat{q}, \delta, \mathbb{A}, \mathbb{G}, \mathbb{P})$  and  $\mathbb{F}' = (X', Y', Q', \hat{q}', \delta', \mathbb{A}', \mathbb{G}', \mathbb{P}')$  are stateful interfaces, and  $\mathbf{f} = (X, Y, Q_{\mathbf{f}}, \hat{q}_{\mathbf{f}}, \delta_{\mathbf{f}}, \mathbb{M})$  and  $\mathbf{f}_{\mathcal{E}} = (Y, X, Q_{\mathcal{E}}, \hat{q}_{\mathcal{E}}, \delta_{\mathcal{E}}, \mathbb{E})$  are stateful components.

A stateful component  $\mathbf{f}$  implements a stateful interface  $\mathbb{F}$  if there exists a simulation relation from  $\mathbf{f}$  to  $\mathbb{F}$  such that the stateless components in the relation implement the stateless interfaces they are related to. Admissible environments require a simulation relation from them to the interface they are admissible on.

**Definition 11.** *A component  $\mathbf{f}$  implements the interface  $\mathbb{F}$ , denoted by  $\mathbf{f} \models \mathbb{F}$ , iff there exists  $H \subseteq Q_{\mathbf{f}} \times Q$  s.t.  $(\hat{q}_{\mathbf{f}}, \hat{q}) \in H$  and for all  $(q_{\mathbf{f}}, q) \in H$ : (i)  $\mathbf{f}(q_{\mathbf{f}}) \models \mathbb{F}(q)$ ; and (ii) if  $q'_{\mathbf{f}} \in \delta_{\mathbf{f}}(q_{\mathbf{f}})$ , then there exists a state  $q' \in \delta(q)$  s.t.  $(q'_{\mathbf{f}}, q') \in H$ . A component  $\mathbf{f}_{\mathcal{E}}$  is an admissible environment for the interface  $\mathbb{F}$ , denoted by  $\mathbf{f}_{\mathcal{E}} \models \mathbb{F}$ , iff there exists a relation  $H \subseteq Q \times Q_{\mathcal{E}}$  s.t.  $(\hat{q}, \hat{q}_{\mathcal{E}}) \in H$  and for all*

$(q, q_{\mathcal{E}}) \in H$ : (i)  $\mathbf{f}(q_{\mathcal{E}}) \models \mathbb{F}(q)$ ; and (ii) if  $q' \in \delta_{\mathbb{F}}(q)$ , then there exists a state  $q'_{\mathcal{E}} \in \delta_{\mathcal{E}}(q_{\mathcal{E}})$  s.t.  $(q', q'_{\mathcal{E}}) \in H$ .

As for stateless interfaces, we have that interface's properties are satisfied after we compose any of its implementations  $\mathbf{f}$  with any of its admissible environments  $\mathbf{f}_{\mathcal{E}}$ .

**Proposition 3.** *For all well-formed interfaces  $\mathbb{F}$ , and all relations  $H \subseteq Q_{\mathbf{f}} \times Q$  and  $H_{\mathcal{E}} \subseteq Q \times Q_{\mathcal{E}}$  that witness  $\mathbf{f} \models \mathbb{F}$  and  $\mathbf{f}_{\mathcal{E}} \models \mathbb{F}$ , respectively, it holds: (i)  $(\mathbb{M}(\hat{q}_{\mathbf{f}}) \cup \mathbb{E}(\hat{q}_{\mathcal{E}}))^* \cap \mathbb{P}(\hat{q}) = \emptyset$ ; and (ii) for all  $q \in Q$  that are reachable from  $\hat{q}$ , if  $(q_{\mathbf{f}}, q) \in H$  and  $(q, q_{\mathcal{E}}) \in H_{\mathcal{E}}$ , then  $(\mathbb{M}(q_{\mathbf{f}}) \cup \mathbb{E}(q_{\mathcal{E}}))^* \cap \mathbb{P}(q) = \emptyset$ .*

Composition of two components is defined as their synchronous product. The composition of two interfaces is defined as their synchronous product, as well. However, we only keep the states that are defined by the composition of two compatible stateless interfaces.

**Definition 12.** *Let  $\mathbb{F}$  and  $\mathbb{F}'$  be two interfaces. Their composition is defined as the tuple:  $\mathbb{F} \otimes \mathbb{F}' = (X_{\mathbb{F}, \mathbb{F}'}, Y_{\mathbb{F}, \mathbb{F}'}, Q_{\mathbb{F}, \mathbb{F}'}, \hat{q}_{\mathbb{F}, \mathbb{F}'}, \delta_{\mathbb{F}, \mathbb{F}'}, \mathbb{A}_{\mathbb{F}, \mathbb{F}'}, \mathbb{G}_{\mathbb{F}, \mathbb{F}'}, \mathbb{P}_{\mathbb{F}, \mathbb{F}'})$ , where:  $\hat{q}_{\mathbb{F}, \mathbb{F}'} = (\hat{q}, \hat{q}')$  and  $Q_{\mathbb{F}, \mathbb{F}'} = \{\hat{q}_{\mathbb{F}, \mathbb{F}'}\} \cup \{(q, q') \mid \mathbb{F}(q) \sim \mathbb{F}'(q')\}$ ;  $(q_2, q'_2) \in \delta_{\mathbb{F}, \mathbb{F}'}(q_1, q'_1)$  iff  $q_2 \in \delta(q_1)$  and  $q'_2 \in \delta'(q'_1)$ ; for all  $(q, q') \in Q_{\mathbb{F}, \mathbb{F}'}$ :  $\mathbb{F}_{\mathbb{F}, \mathbb{F}'}(q, q') = \mathbb{F}(q) \otimes \mathbb{F}'(q')$ .*

Two stateful interfaces are compatible if the stateless interfaces defined by their initial states are compatible, i.e.  $\mathbb{F}(\hat{q}) \sim \mathbb{F}'(\hat{q}')$ . It follows from the results proved for the stateless interfaces that compatibility is commutative, composition preserves well-formedness and stateful interfaces support incremental design.

**Proposition 4.** *If  $\mathbf{f} \models \mathbb{F}$  and  $\mathbf{g} \models \mathbb{G}$ , then  $\mathbf{f} \otimes \mathbf{g} \models \mathbb{F} \otimes \mathbb{G}$ .*

Given an interface, we define transitions parameterized by no-flows on its input variables (i.e. with fixed assumptions) or on its output variables (i.e. with fixed guarantees and properties).

**Definition 13.** *Let  $\mathbb{F}$  be an interface. Input transitions from a given state  $q \in Q$  are defined as  $\delta^X(q) = \{\delta^X(q, \mathcal{A}) \mid \mathcal{A} \subseteq Z \times X\}$  with  $\delta^X(q, \mathcal{A}) = \{q' \in \delta(q) \mid \mathbb{A}(q') = \mathcal{A}\}$ . Output transitions from a given state  $q \in Q$  are defined as  $\delta^Y(q) = \{\delta^Y(q, \mathcal{G}, \mathcal{P}) \mid \mathcal{G} \subseteq Z \times Y \text{ and } \mathcal{P} \subseteq Z \times Y\}$  with  $\delta^Y(q, \mathcal{G}, \mathcal{P}) = \{q' \in \delta(q) \mid \mathbb{G}(q') = \mathcal{G} \text{ and } \mathbb{P}(q') = \mathcal{P}\}$ .*

Interface  $\mathbb{F}_R$  refines  $\mathbb{F}_A$ , if all output steps of  $\mathbb{F}_R$  can be simulated by  $\mathbb{F}_A$ , while all input steps of  $\mathbb{F}_A$  can be simulated by  $\mathbb{F}_R$ . This corresponds to alternating refinement [5].

**Definition 14.** *Interface  $\mathbb{F}_R$  refines  $\mathbb{F}_A$ , written  $\mathbb{F}_R \preceq \mathbb{F}_A$ , iff there exists a relation  $H \subseteq Q_R \times Q_A$  s.t.  $(\hat{q}_R, \hat{q}_A) \in H$  and for all  $(q_R, q_A) \in H$ : (i)  $\mathbb{F}_R(q_R) \preceq \mathbb{F}_A(q_A)$ ; (ii) for all set of states  $O \in \delta_R^Y(q_R)$ , there exists  $O' \in \delta_A^Y(q_A)$  s.t. for all set of states  $I' \in \delta_A^X(q_A)$ , there exists  $I \in \delta_R^X(q_R)$  s.t.  $(O \cap I) \times (O' \cap I') \subseteq H$ .*

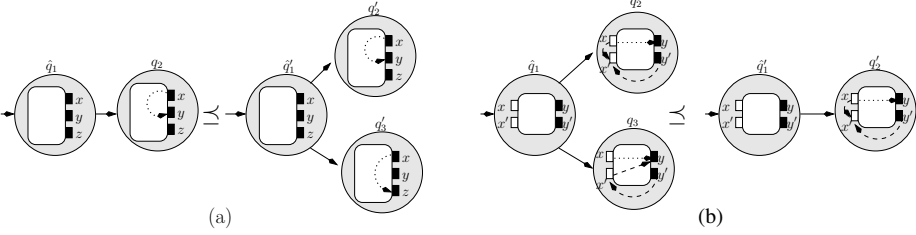


Fig. 8: Refined interfaces with witness: (a) relation  $\{(\hat{q}_1, \hat{q}'_1), (q_2, q'_2)\}$ ; and (b) relation  $\{(\hat{q}_1, \hat{q}'_1), (q_2, q'_2), (q_3, q'_2)\}$ .

*Example 5.* In Figure 8 we depict two examples of refined stateful interfaces.

In Figure 8(a) the stateless interface in each state only uses output ports and it only specifies properties. The initial state of both stateful interfaces is the same, so they clearly refine each other. As there are no assumptions and guarantees, then, by Definition 14, we need to check that for all successors of the initial state in the refined interface  $q_s$ , there exists a successor of the initial state in the abstract interface  $q'_s$  such that  $\mathbb{P}_A(q'_s) \subseteq \mathbb{P}_R(q_s)$ . This holds for the states  $(q_2, q'_2)$ . Hence the relation  $\{(\hat{q}_1, \hat{q}'_1), (q_2, q'_2)\}$  witnesses the refinement. Note that the refined interface is obtained by removing a nondeterministic choice on the transition function.

The witness relation for the refinement depicted in Figure 8(b) is  $\{(\hat{q}_1, \hat{q}'_1), (q_2, q'_2), (q_3, q'_2)\}$ . The initial states are the same, so the condition (i) in Definition 14 is trivially satisfied. The refined interface has two distinct *output transitions* from the initial state  $\hat{q}_1$ . It can either go to state  $q_2$  by choosing the set of guarantees and proposition with only one element  $(x, y)$  or it can transition to state  $q_3$  by committing to the set of no-flows  $\{(x, y), (x', y)\}$  for the guarantees and  $\{(x, y)\}$  as property. From the initial state of the abstract interface, there exists only one *input transition* possible, to assume that  $x$  does not flow to  $x'$  and  $y'$  does not flow to  $x$ . The following holds for both states accessible from the initial state in the refined interface:  $\mathbb{A}_R(q_2) \subseteq \mathbb{A}_A(q'_2)$  and  $\mathbb{A}_R(q_3) \subseteq \mathbb{A}_A(q'_2)$ . The refined interface specifies an alternative transition from the initial state (represented by state  $q_3$ ) that allows more environments while restricting the implementation and preserving the property.

**Theorem 5.** *Let  $\mathbb{F}' \preceq \mathbb{F}$ . (a) If  $\mathbf{f} \models \mathbb{F}'$ , then  $\mathbf{f} \models \mathbb{F}$ . (b) If  $\mathbf{f}_\mathcal{E} \models \mathbb{F}$ , then  $\mathbf{f}_\mathcal{E} \models \mathbb{F}'$ .*

**Theorem 6 (Independent implementability).** *For all well-formed interfaces  $\mathbb{F}'_1, \mathbb{F}_1$  and  $\mathbb{F}_2$ , if  $\mathbb{F}'_1 \preceq \mathbb{F}_1$  and  $\mathbb{F}_1 \sim \mathbb{F}_2$ , then  $\mathbb{F}'_1 \sim \mathbb{F}_2$  and  $\mathbb{F}'_1 \otimes \mathbb{F}_2 \preceq \mathbb{F}_1 \otimes \mathbb{F}_2$ .*

The composition operation on stateful information-flow interfaces can be generalized to distinguish between compatible and incompatible transitions of interfaces when they are composed. Usually this is done by labeling transitions with letters from an alphabet, so that only transitions with the same letter can be synchronized. While necessary for practical modeling, we omit this technical generalization to allow the reader to focus on the novelty of our formalism, which

is the ability to specify information-flow constraints (environment assumptions, implementation guarantees, and global properties) at each state of an interface.

## 5 Related Work

To the best of our knowledge, we are the first to provide a theory for top-down and bottom-up design of information-flow system requirements that supports both incremental design and independent implementability of systems. The literature closest to our work about information-flow focus on the semantic aspects of it. The novelty of our work lies on explicit separation of the structural concerns from the semantic aspects of information-flow.

Language-based techniques have been proved useful to verify and enforce information flow policies [29]. Examples range from type systems [15] to program analysis using program-dependency graphs (PDGs) [18,16]. In our approach we aim at composition and refinement notions that are independent of the language adopted for the implementations.

Information-flow properties can be specified with respect to the observed behavior of a system, in which each of its execution runs is abstracted as a trace. In this approach, properties often compare multiple executions of a system to certify that no forbidden flow can be deduced by an observer. Such properties over multiple execution traces are called hyperproperties [12]. Temporal logics [26], like LTL or CTL\* are used to specify trace properties of reactive systems. HyperLTL and HyperCTL\* [11] extend temporal logics by introducing quantifiers over path variables. They allow relating multiple executions and expressing information-flow security properties [12,11]. Epistemic temporal logics (ETL) [9] provide the knowledge connective with an implicit quantification over traces. With ETL we can reason about the knowledge gain of agents over time. Then, we can specify which information can be learned by the agents while interacting with the system [6]. All these LTL extensions reason about closed systems while our approach allows compositional reasoning about open systems. Moreover, we focus here on the structural aspect of information-flow, and not yet on its semantic interpretation. Thus, all information-flow trace-based semantics are orthogonal to our approach.

Interface theories belong to the broader area of contract-based design [8], originally popularized by Meyer [24], following earlier ideas introduced by Floyd and Hoare [14,19]. Our theory follows closely the philosophy for formal frameworks for systems design introduced for Interface automata (IA) [1] and Assume/Guarantee (A/G) [2] interfaces. Interface theories were later extended with extra-functional requirements such as resource [10], timing [4,13] and security [21] requirements. Unlike in previous interface formalisms, we had to introduce the notion of *properties* which capture the intent of the designer and can be used to steer the refinement of interfaces.

*Interface for structure and security* (ISS) [21] is a variant of IA that enables specification of two types of actions on (1) low and (2) high confidential information. ISS uses a bisimulation-based notion of non-interference that checks



whether the system behaves in the same way when high actions are performed or when they are considered hidden actions. Our approach is orthogonal to IA and their extensions: we do not characterise the type of actions of each component, but only their input/output ports, defining explicitly the information-flow relations between variables.

Our approach took inspiration from *relational interfaces* (RIs) [31]. RIs specify the legal inputs that the environment is allowed to provide to the component along with the legal outputs that the component can generate when provided with these input. RIs do not have assumptions and guarantees defined separately. Instead, they have a contract that specifies the desired input-output behavior. A contract in RIs is expressed over individual traces. Then, an RI contract can only relate input and output values in a trace, and not across multiple traces. This restricts considerably RIs expressivity concerning information-flow properties. Besides, RIs are trace-based interfaces, while in our approach we focus on the structural aspect of information-flow, which may change from state to state (in the stateful case). Our approach can be seen as a limited way to introduce relational properties into A/G interfaces, namely solely for guiding refinement. This limited way avoids many of the technical complexities of general relational interfaces [31].

## 6 Conclusion

We propose a novel interface theory to specify information-flow properties. Our framework includes both *stateless* and *stateful* interfaces and supports both incremental design and independent implementability. To achieve this, unlike in previous interface formalisms, we introduce the notion of *properties* which captures the intent of the designer for the interaction between assumptions and guarantees. Moreover, properties can be used to steer the refinement of interfaces. It will be interesting to study the introduction of such design-guiding properties in the context of other interface languages.

As future work, we will explore how to extend our theory with sets of *must-flows*, i.e. support for modal specifications [27]. This will enable, for example, to specify flows that a state  $q$  must implement so that the system can transition to a different state, which is useful to specify declassification of information. Another direction is to explore trace semantics for our interfaces.

## References

1. de Alfaro, L., Henzinger, T.A.: Interface automata. In: European Software Engineering Conference/Foundations on Software Engineering (ESEC/FSE). p. 109120. ACM (2001). <https://doi.org/10.1145/503209.503226>
2. de Alfaro, L., Henzinger, T.A.: Interface theories for component-based design. In: Embedded Software. LNCS, vol. 2211, pp. 148–165. Springer (2001). [https://doi.org/10.1007/3-540-45449-7\\_11](https://doi.org/10.1007/3-540-45449-7_11)

3. de Alfaro, L., Henzinger, T.A.: Interface-based design. In: Engineering Theories of Software Intensive Systems. NATO Science Series (Series II: Mathematics, Physics and Chemistry), vol. 195, pp. 83–104. Springer Netherlands (2005). [https://doi.org/10.1007/1-4020-3532-2\\_3](https://doi.org/10.1007/1-4020-3532-2_3)
4. de Alfaro, L., Henzinger, T.A., Stoelinga, M.: Timed interfaces. In: Embedded Software. LNCS, vol. 2491, pp. 108–122. Springer (2002). [https://doi.org/10.1007/3-540-45828-X\\_9](https://doi.org/10.1007/3-540-45828-X_9)
5. Alur, R., Henzinger, T.A., Kupferman, O., Vardi, M.Y.: Alternating refinement relations. In: CONCUR'98 Concurrency Theory. LNCS, vol. 1466, pp. 163–178. Springer (1998). <https://doi.org/10.1007/BFb0055622>
6. Balliu, M., Dam, M., Le Guernic, G.: Epistemic temporal logic for information flow security. In: Proceedings of the ACM SIGPLAN 6th Workshop on Programming Languages and Analysis for Security (PLAS). pp. 1–12. ACM (2011). <https://doi.org/10.1145/2166956.2166962>
7. Benadjila, R., Renard, M., Lopes-Esteves, J., Kasmı, C.: One car, two frames: attacks on hitag-2 remote keyless entry systems revisited. In: 11th USENIX Workshop on Offensive Technologies (2017)
8. Benveniste, A., Caillaud, B., Nickovic, D., Passerone, R., Raclet, J., Reinkemeier, P., Sangiovanni-Vincentelli, A.L., Damm, W., Henzinger, T.A., Larsen, K.G.: Contracts for system design. Foundations and Trends in Electronic Design Automation **12**(2-3), 124–400 (2018). <https://doi.org/10.1561/10000000053>
9. Bozzelli, L., Maubert, B., Pinchinat, S.: Unifying hyper and epistemic temporal logics. In: Foundations of Software Science and Computation Structures (FoSSaCS). LNCS, vol. 9034, pp. 167–182. Springer (2015). [https://doi.org/10.1007/978-3-662-46678-0\\_11](https://doi.org/10.1007/978-3-662-46678-0_11)
10. Chakrabarti, A., de Alfaro, L., Henzinger, T.A., Stoelinga, M.: Resource interfaces. In: Embedded Software. LNCS, vol. 2855, pp. 117–133. Springer (2003). [https://doi.org/10.1007/978-3-540-45212-6\\_9](https://doi.org/10.1007/978-3-540-45212-6_9)
11. Clarkson, M.R., Finkbeiner, B., Koleini, M., Micinski, K.K., Rabe, M.N., Sánchez, C.: Temporal logics for hyperproperties. In: Principles of Security and Trust (POST). LNCS, vol. 8414, pp. 265–284. Springer (2014). [https://doi.org/10.1007/978-3-642-54792-8\\_15](https://doi.org/10.1007/978-3-642-54792-8_15)
12. Clarkson, M.R., Schneider, F.B.: Hyperproperties. Journal of Computer Security **18**(6), 1157–1210 (2010). <https://doi.org/10.3233/JCS-2009-0393>
13. David, A., Larsen, K.G., Legay, A., Nyman, U., Wasowski, A.: 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). pp. 91–100. ACM (2010). <https://doi.org/10.1145/1755952.1755967>
14. Floyd, R.W.: Assigning meanings to programs. Proceedings of Symposium on Applied Mathematics **19**, 19–32 (1967). [https://doi.org/10.1007/978-94-011-1793-7\\_4](https://doi.org/10.1007/978-94-011-1793-7_4)
15. Focardi, R., Maffei, M.: Types for security protocols. Formal Models and Techniques for Analyzing Security Protocols **5**, 143–181 (2011). <https://doi.org/10.3233/978-1-60750-714-7-143>
16. Graf, J., Hecker, M., Mohr, M.: Using JOANA for information flow control in Java programs - a practical guide. In: Software Engineering 2013 - Workshopband. LNI, vol. P-215, pp. 123–138. Gesellschaft für Informatik e.V. (2013), <https://dl.gi.de/20.500.12116/17361>
17. Hamilton, M.D., Tunstall, M., Popovici, E.M., Marnane, W.P.: Side channel analysis of an automotive microprocessor. In: IET Irish Signals and Systems

- Conference (ISSC). pp. 4–9. Institution of Engineering and Technology (2008). <https://doi.org/10.1049/cp:20080630>
18. Hammer, C., Snelling, G.: Flow-sensitive, context-sensitive, and object-sensitive information flow control based on program dependence graphs. *International Journal of Information Security* **8**(6), 399–422 (2009). <https://doi.org/10.1007/s10207-009-0086-1>
  19. Hoare, C.A.R.: An axiomatic basis for computer programming. *Communications of the ACM* **12**(10), 576–580 (1969). <https://doi.org/10.1145/363235.363259>
  20. Larsen, K.G., Nyman, U., Wasowski, A.: Interface input/output automata. In: Misra, J., Nipkow, T., Sekerinski, E. (eds.) *International Symposium on Formal Methods (FM)*. LNCS, vol. 4085, pp. 82–97. Springer (2006). [https://doi.org/10.1007/11813040\\_7](https://doi.org/10.1007/11813040_7)
  21. Lee, M., D’Argenio, P.R.: Describing secure interfaces with interface automata. *Electronic Notes in Theoretical Computer Science* **264**(1), 107–123 (2010). <https://doi.org/10.1016/j.entcs.2010.07.008>
  22. Mantel, H.: On the composition of secure systems. In: *IEEE Symposium on Security and Privacy*. pp. 88–101. IEEE Computer Society (2002). <https://doi.org/10.1109/SECPR.2002.1004364>
  23. Mantel, H., Sands, D., Sudbrock, H.: Assumptions and guarantees for compositional noninterference. In: *IEEE Computer Security Foundations Symposium (CSF)*. pp. 218–232. IEEE (2011). <https://doi.org/10.1109/CSF.2011.22>
  24. Meyer, B.: Applying ‘design by contract’. *Computer* **25**(10), 40–51 (1992). <https://doi.org/10.1109/2.161279>
  25. Mikulcak, M., Herber, P., Göthel, T., Glesner, S.: Information flow analysis of combined simulink/stateflow models. *Information Technology And Control* **48**(2), 299–315 (2019). <https://doi.org/10.5755/j01.itc.48.2.21759>
  26. Pnueli, A.: The temporal logic of programs. In: *Annual Symposium on Foundations of Computer Science (FOCS)*. pp. 46–57. IEEE Computer Society (1977). <https://doi.org/10.1109/SFCS.1977.32>
  27. Racllet, J.B., Badouel, E., Benveniste, A., Caillaud, B., Legay, A., Passerone, R.: A modal interface theory for component-based design. *Fundamenta Informaticae* **108**(1-2), 119–149 (2011). <https://doi.org/10.3233/FI-2011-416>
  28. Ratasich, D., Khalid, F., Geissler, F., Grosu, R., Shafique, M., Bartocci, E.: A roadmap toward the resilient internet of things for cyber-physical systems. *IEEE Access* **7**, 13260–13283 (2019). <https://doi.org/10.1109/ACCESS.2019.2891969>
  29. Sabelfeld, A., Myers, A.C.: Language-based information-flow security. *IEEE Journal on Selected Areas in Communications* **21**(1), 5–19 (2003). <https://doi.org/10.1109/JSAC.2002.806121>
  30. Schneider, F.B.: Enforceable security policies. *ACM Transactions on Information and System Security* **3**(1), 30–50 (2000). <https://doi.org/10.1145/353323.353382>
  31. Tripakis, S., Lickly, B., Henzinger, T.A., Lee, E.A.: A theory of synchronous relational interfaces. *ACM Transactions on Programming Languages and Systems (TOPLAS)* **33**(4), 14 (2011). <https://doi.org/10.1145/1985342.1985345>
  32. Verdult, R., Garcia, F.D., Balasch, J.: Gone in 360 seconds: Hijacking with hitag2. In: *21st USENIX Security Symposium*. pp. 237–252 (2012)

**Open Access** This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (<http://creativecommons.org/licenses/by/4.0/>), which permits use, sharing, adaptation, distribution and reproduction in any medium

or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.

