Centre Fédéré en Vérification

Technical Report number 2005.52

Sociable Interfaces

Luca de Alfaro, Leandro Dias da Silva, Marco Faella, Axel Legay, Pritam Roy, Maria Sorea

This work was partially supported by a FRFC grant: 2.4530.02

http://www.ulb.ac.be/di/ssd/cfv

Sociable Interfaces*

Luca de Alfaro[1], Leandro Dias da Silva[1,2], Marco Faella[1,3], Axel Legay[1,4], Pritam Roy[1], and Maria Sorea[5]

[1] School of Engineering, Universitity of California, Santa Cruz, USA
[2] Electrical Engineering Department, Federal University of Campina Grande, Paraiba, Brazil
[3] Dipartimento di Scienze Fisiche, Università di Napoli "Federico II", Italy
[4] Department of Computer Science, University of Liège, Belgium
[5] School of Computer Science, University of Manchester, United Kingdom

Abstract. Interface formalisms are able to model both the input requirements and the output behavior of system components; they support both bottom-up component-based design, and top-down design refinement. In this paper, we propose "sociable" interface formalisms, endowed with a rich compositional semantics that facilitates their use in design and modeling. Specifically, we introduce interface models that can communicate via both actions and shared variables, and where communication and synchronization covers the full spectrum, from one-to-one, to one-to-many, many-to-one, and many-to-many. Thanks to the expressive power of interface formalisms, this rich compositional semantics can be realized in an economical way, on the basis of a few basic principles. We show how the algorithms for composing, checking the compatibility, and refining the resulting sociable interfaces can be implemented symbolically, leading to efficient implementations.

1 Introduction

Interface theories are formal models of communicating systems. Compared to traditional models, the strength of interface theories lies in their ability to model both the input requirements, and the output behavior, of a system. This gives rise to a *compatibility* test when interface models are composed: two interfaces are compatible if there is a way to use them (an environment) in which their input assumptions are simultaneously satisfied. This ability to model input assumptions and provide a compatibility test makes interface models useful in system design. In particular, interface models support both bottom-up, and top-down, design processes [6, 7]. In a bottom-up direction, the compatibility test can be used to check that portions of the design work correctly, even before all the components are assembled in the final design. In a top-down direction, interface models enable the hierarchical decomposition of a design specification, while providing a guarantee that if the components satisfy their specifications, then they will interact correctly in the overall implementation.

* This research was supported in part by the NSF CAREER award CCR-0132780, by the ONR grant N00014-02-1-0671, by the ARP award TO.030.MM.D., by awards from the Brazilian government agencies CNPq and CAPES, and by a F.R.I.A Grant

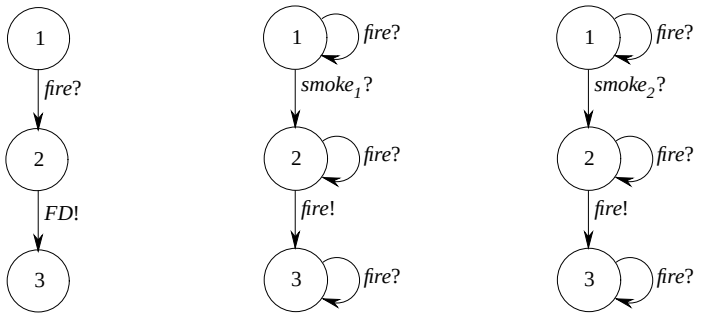
In this paper we present interfaces models that can communicate via both actions and variables, and that provide one-to-one, many-to-one, one-to-many, and many-to-many communication and synchronization. We show that this rich communication semantics can be achieved by combining a small number of basic concepts, thanks to the expressive power of interface models. This leads to an uniform, and conceptually simple, communication model. We call this model *sociable interfaces*, underlining the ease with which these interfaces can be composed into models of design. While sociable interfaces do not break new ground in the conceptual theory of interface models, we hope that they constitute a useful step towards a practical, interface-based design methodology.

In sociable interfaces, synchronization and communication are based on two main ideas. The first idea is that the same action can appear as a label of both input and output transitions: when the action labels output transitions, it means that the interface can emit the action; when the action labels an input transition, it means that the action can be accepted if sent from other components. Depending on whether the action labels only input transitions, only output transitions, or both kind of transitions, we have different synchronization schemes. For instance, if an action a is associated only with output transitions, it means that the interface can emit a, but cannot receive it, and thus it cannot be composed with any other interface that emits a. Conversely, if a is associated only with input transitions, it means that the interface accepts a from other interfaces, but will not emit a. Finally, if a is associated both with input and output transitions, it means that the interface can both emit a, and accept a when emitted by other interfaces.

The second idea is that global variables do not belong to specific interfaces: the same global variable can be updated by multiple interfaces. In an interface, the output transitions associated with an action specifies how global variables can be updated when the interface emits the action; the input transition associated with an action specifies constraints on how other interfaces can update the global variables when emitting the action. By limiting the sets of variables whose value must be tracked by the interfaces, and by introducing appropriate non-interference conditions among interfaces, we can ensure that interfaces can participate in complex communication schemes with limited knowledge about the other participants. In particular, interfaces do not need to know in advance the number or identities of the other interfaces that take part in communication schemes. This facilitates component reuse, as the same interface model can be used in different contexts.

We show that the compatibility and refinement of sociable interfaces can be checked via efficient symbolic algorithms. We have implemented these algorithms in a tool called TIC (Tool for Interface Compatibility); the tool is written in Ocaml [10], and the symbolic algorithms for interface compatibility and refinement are built on top of the MDD/BDD Glue and Cudd packages [13, 12].

The paper is organized as follows. First, we introduce *sociable interface automata*, which include actions, but not variables, and which are a more "sociable" version of the interface automata of [6, 8]. After illustrating the various synchronization and communication features for sociable interface automata, we endow them with variables in Section 3, obtaining *sociable interface modules*. We describe the communication mechanisms of sociable interface modules via examples, and we show how the examples can



(a) C: Control Unit (b) D_1: Fire Detector 1 (c) D_2: Fire Detector 2

Fig. 1. Sociable interface automata for a fire detection and reporting system.

be encoded in the input language of the tool TIC. The refinement of sociable interfaces is discussed Section 4, and the symbolic implementation of the composition and refinement algorithms is in Section 5. We conclude with a comparison between sociable interfaces and previous interface formalisms.

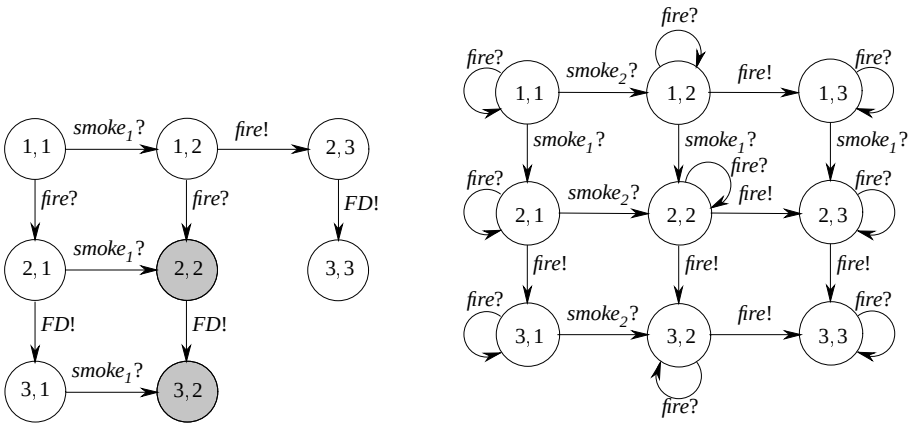
2 Sociable Interface Automata

Social interfaces communicate via both actions and variables. We first illustrate how sociable interfaces communicate via actions; in the next section, we will augment them with variables, obtaining the model implemented in the tool TIC. We begin with an informal, intuitive preview, which will motivate the definitions.

2.1 Preview

To provide some intuition on sociable interfaces, we present an example: a very simple model of a fire detection and reporting system. The sociable interfaces for this example are depicted in Figure 1: D_1 and D_2 are the fire detectors (there could be more), and C is the control unit. When the fire detectors D_1 and D_2 detect smoke (input events $smoke_1?$,$smoke_2?$), they generate an output event *fire!*. The control unit, upon receiving the input event *fire?*, issues a call for the fire department (output event *FD!*). Similar to the original interface model [6, 8], the input and output transitions departing from a state of a sociable interface denote the inputs that can be received, and the outputs that can be generated, from that state. For instance, the sociable interface C (Figure 1(a)) specifies that input event *fire?* can be accepted at state 1, but not at state 2.

Product and composition. To compose two sociable interfaces, we first form their automata product. In the product, shared output/input events (such as the pair *fire!*–*fire?* in Figure 1) synchronize: this models communication, or synchronization, initiated by the



(a) $C \otimes D_1$ (b) $D_1 \otimes D_2$

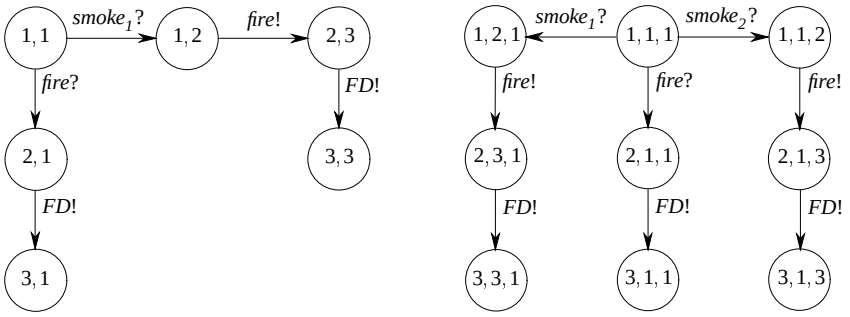
Fig. 2. Product of the automata D_1, D_2, and C.

interface issuing the output transition. Similarly, two interfaces can also synchronize on shared inputs: when the environment generates an input, both interfaces will receive it and take the corresponding input transition. However, interfaces do not synchronize on shared outputs: as an example, D_1 and D_2 do not synchronize on the output event *fi re!* in their product $D_1 \otimes D_2$ (Figure 2(b)). The idea is that, in an asynchronous model, independent components issue their output asynchronously, so that synchronization cannot happen. As usual, interfaces do not synchronize on non-shared actions.

In the product of two interfaces, we distinguish between *good* and *bad* states. A state is *good* if all the outputs produced by one component can be accepted as inputs by the other component; a state is bad otherwise. For instance, in the product $C \otimes D_1$ (Figure 2(a)), the states $\langle 2,2 \rangle$ and $\langle 3,2 \rangle$ are *bad*, since from state 2 the detector D_1 can issue *fi re!*, and this cannot be matched by an input transition *fi re?* neither from state 2 nor from state 3 of the control unit.

A state of the product is *compatible* if there is an Input strategy that can avoid all bad states: this means that starting from that state, there is an environment under which the component interfaces interact correctly. The composition of two interfaces is obtained by removing all incompatible states from the product. The composition $C \| D_1$ of C and D_1 is depicted in Figure 3(a), and the composition of $C \| D_1 \| D_2$ is depicted in Figure 3(b). Notice that in the composition $C \| D_1 \| D_2$, once $smoke_1$ (resp. $smoke_2$) is received, $smoke_2$ (resp. $smoke_1$) is not allowed. This behavior results from the design of the control unit which cannot accept more than one "smoke-input" before issuing *FD!*.

Multi-way communication. In a sociable interface, the same action can label both input and output transitions: this is illustrated, for instance, by action *fi re* in Figures 1(b) and 1(c). Indeed, sociable interfaces do not have separate input and output transition



(a) $C \| D_1$ (b) $C \| D_1 \| D_2$

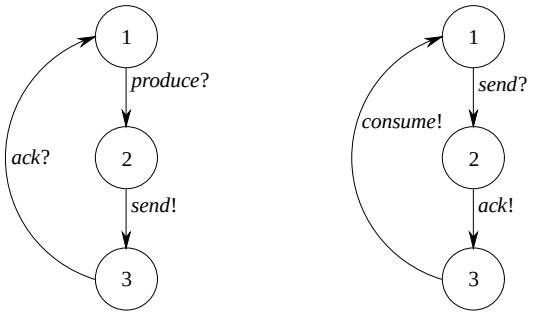
Fig. 3. Composition of the automata D_1, D_2, and C.

alphabets: rather, they have a single *action alphabet*, and actions in this alphabet can label edges both as inputs, giving rise to *input transitions*, and as outputs, giving rise to *output transitions*. For example, the action *fire* at state 2 of D_1 corresponds to both an output, and to an input transition: this indicates that D_1 can generate output *fire*, while at the same time being composable with other interfaces that generate *fire* as output (such as D_2). Thus, if an action a is in the alphabet of an interface, there are four cases:

- If a is not associated with any transition, then the interface neither outputs a, nor can it be composed with other interfaces that do.
- If a is associated with output transitions only, then the interface can generate a, but it cannot be composed with interfaces that also output a.
- If a is associated with input transitions only, then the interface can receive a, but not output it.
- If a is associated with both input and output transitions, then the interface can generate a, and it can be composed with other interfaces that do.

We notice how these four cases all arise in an uniform way from our interpretation of input and output edges. All of these cases have a use in system modeling: the fire detector example illustrated the non-exclusive generation of outputs, the next example illustrates exclusive generation.

Figure 4 depicts a simple communication protocol. In this protocol, the sender *Se*, after receiving information from the environment (label *produce?*), sends this information to the receiver (label *send!*), and awaits for an acknowledge (label *ack?*). The lack of input edges labeled with *send* in *Se*, and the lack of input edges labeled with *ack* in *Re* indicate that the communication channel between *Se* and *Re* is not shared: only *Se* can generate *send* actions, and only *Re* can generate *ack* actions.



(a) *Se*: Sender (b) *Re*: Receiver

Fig. 4. A simple communication protocol.

2.2 Definitions

Given two sets A and B, we denote with $A \rightrightarrows B$ the set of *nondeterministic functions* from A to B, that is: $A \to 2^B$.

Definition 1 (Sociable Interface Automaton). A *sociable interface automaton* (automaton for short) is a tuple $M = (Act, S, \tau^I, \tau^O, \varphi^I, \varphi^O)$, where:

- Act is a set of *actions*.
- S is a set of *states*.
- $\tau^I : Act \times S \rightrightarrows S$ is the *input transition function*.
- $\tau^O : Act \times S \rightrightarrows S$ is the *output transition function*.
- $\varphi^I \subseteq S$ is the *input invariant*.
- $\varphi^O \subseteq S$ is the *output invariant*.

We require τ^I to be deterministic, that is: for all $s \in S$ and $a \in Act$, $|\tau^I(a,s)| \leq 1$.

For all $s \in S$ and $a \in Act$, we define $\widehat{\tau}^I(a,s) = \tau^I(a,s) \cap \varphi^I$, and $\widehat{\tau}^O(a,s) = \tau^O(a,s) \cap \varphi^O$. Together, S, τ^I and τ^O define a graph whose edges are labeled with actions in Act. As it was already informally done in the examples of Section 2.1, we therefore depict interface automata as graphs. To distinguish input from output transitions, we add a tag at the end of the name of the action: as in process algebra notation, we add "?" for input transitions and "!" for output transitions. In all examples, it holds $\varphi^I = \varphi^O = S$.

Example 1. Figure 1(b) is a graphical representation of a 3-state automaton whose actions are *fire*, and *smoke*$_1$. For instance, from state 2, the automaton can take an input transition *fire?*, as well as an output transition *fire!*.

The semantics of a sociable interface automaton can be described in terms of a game between two players, Input and Output, played over the graph representation of the automaton. At each round, from the current state in the graph, the Input player

chooses an outgoing input edge, and the Output player chooses an outgoing output edge. In order to ensure that both players always have an enabled move, we introduce a special move Δ_0 which, when played, gives rise to a *stuttering step*, that is, a step that does not change the current state of the automaton. Furthermore, we postulate that player Output (resp. Input) can choose only edges that lead to states where the output (resp. input) invariant holds. Thus, input and output invariants are used to restrict the set of moves available to the players; their true usefulness will become clearer when considering interfaces with variables, i.e. *modules*.

In the remaining of this section, we consider a fixed sociable interface automaton $M = (Act_M, S_M, \tau_M^I, \tau_M^O, \varphi_M^I, \varphi_M^O)$. The sets of enabled moves can be defined as follows.

Definition 2 (Moves). For all $s \in S_M$, the set of moves for player Input at s is given by:

$$\Gamma^I(M,s) = \{\Delta_0\} \cup \{\langle a, s'\rangle \in Act_M \times S_M \mid s' \in \widehat{\tau}_M^I(a,s)\}.$$

Similarly, the set of moves for player Output at s is given by:

$$\Gamma^O(M,s) = \{\Delta_0\} \cup \{\langle a, s'\rangle \in Act_M \times S_M \mid s' \in \widehat{\tau}_M^O(a,s)\}.$$

Example 2. Consider the automaton D_1 of Example 1, we have that $\Gamma^I(D_1,1) = \{\Delta_0, \langle \mathit{fire}, 1\rangle, \langle \mathit{smoke}_1, 2\rangle\}$, and $\Gamma^O(D_1,2) = \{\Delta_0, \langle \mathit{fire}, 3\rangle\}$.

At each game round, both players choose a move from the corresponding set of enabled moves. The outcome of their choice is defined as follows.

Definition 3 (Move Outcome). For all states $s \in S_M$ and moves $m^I \in \Gamma^I(M,s)$ and $m^O \in \Gamma^O(M,s)$, the *outcome* $\delta(M,s,a^I,a^O) \in S_M$ of playing m^I and m^O at s can be defined as follows, according to whether m^I and m^O are Δ_0 or a move of the form $\langle a, s'\rangle$.

$$\delta(M,s,\Delta_0,\Delta_0) = \{s\}, \qquad \delta(M,s,\Delta_0,\langle a,s'\rangle) = \{s'\},$$
$$\delta(M,s,\langle a,s'\rangle,\Delta_0) = \{s'\}, \qquad \delta(M,s,\langle a,s'\rangle,\langle b,t'\rangle) = \{s',t'\}.$$

A *strategy* represents the behavior of a player in the game. A strategy is a function that, given the history of the game, i.e., the sequence of states visited in the course of the game, yields one of the player's enabled moves.

For $s \in S_M$, we define the set of *finite runs* starting from s as the set $Runs(M,s) \subseteq S_M^*$ of all finite sequences $s_0 s_1 s_2 \ldots s_n$, such that $s_0 = s$, and for all $0 \leq i < n$, $s_{i+1} \in \delta(M, s_i, m^I, m^O)$, for some $m^I \in \Gamma^I(M,s_i)$, $m^O \in \Gamma^O(M,s_i)$. We also set $Runs(M) = \bigcup_{s \in S_M} Runs(M,s)$.

Definition 4 (Strategy). A *strategy* for player $p \in \{I,O\}$ in an automaton M is a function $\pi^p : Runs(M) \to Act_M \cup \{\Delta_0\}$ that associates, with every run $\sigma \in Runs(M)$ whose final state is s, a move $\pi^p(\sigma) \in \Gamma^p(M,s)$. We denote by Π_M^I and Π_M^O the set of input and output strategies for M, respectively.

An input and an output strategy jointly determine a *set* of *outcomes* in $Runs(M)$.

Definition 5 (Strategy Outcome). Given a state $s \in S_M$, an input strategy $\pi^I \in \Pi^I_M$ and an output strategy $\pi^O \in \Pi^O_M$, the set *outcomes* $\hat{\delta}(M,s,\pi^I,\pi^O)$ of π^I and π^O from s consists of all finite runs $\sigma = s_0 s_1 s_2 \dots s_n$ such that $s = s_0$, and for all $0 \leq i < n$, $s_{i+1} \in \delta(M, s_i, \pi^I(\sigma_{0:i}), \pi^O(\sigma_{0:i}))$, where $\sigma_{0:i}$ denotes the prefix $s_0 s_1 s_2 \dots s_i$ of σ.

Definition 6 (Winning States). Given a state $s \in S_M$ and a goal $\gamma \subseteq \mathit{Runs}(M,s)$, we say that s is *winning* for input with respect to γ, and we write $s \in \mathit{Win}^I(M,\gamma)$, iff there is $\pi^I \in \Pi^I_M$ such that for all $\pi^O \in \Pi^O_M$, $\hat{\delta}(M,s,\pi^I,\pi^O) \subseteq \gamma$. Similarly, we say that s is *winning* for output with respect to γ, and we write $s \in \mathit{Win}^O(M,\gamma)$, iff there is $\pi^O \in \Pi^O_M$ such that for all $\pi^I \in \Pi^I_M$, $\hat{\delta}(M,s,\pi^I,\pi^O) \subseteq \gamma$.

A state of an automaton is *well-formed* if both players have a strategy to always satisfy their own invariant. Following temporal logic notation, for all $X \subseteq S_M$, we denote by $\Box X$ the set of all runs in $\mathit{Runs}(M)$ all whose states belong to X. Formally, $\Box X = \{s_0 s_1 s_2 \dots s_n \in \mathit{Runs}(M) \mid \forall 0 \leq i \leq n \,.\, s_i \in X\}$.

Definition 7 (Well-formed State). We say that a state $s \in S_M$ is *well-formed* iff $s \in \mathit{Win}^I(M, \Box\varphi^I_M) \cap \mathit{Win}^O(M, \Box\varphi^O_M)$.

Notice that if s is well-formed, then $s \in \varphi^I_M \cap \varphi^O_M$.

Definition 8 (Normal Form). We say that M is in *normal form* iff $\varphi^I_M = \mathit{Win}^I(M, \Box\varphi^I_M)$, and $\varphi^O_M = \mathit{Win}^O(M, \Box\varphi^O_M)$.

Given an automaton M_1, we can define an automaton M_2 such that the well-formed portion of M_1 coincides with the one of M_2, and M_2 is in normal form. Let $M_1 = (\mathit{Act}_1, S_1, \tau^I_1, \tau^O_1, \varphi^I_1, \varphi^O_1)$, we set $M_2 = (\mathit{Act}_1, S_1, \tau^I_2, \tau^O_2, \varphi^I_2, \varphi^O_2)$, where, $\varphi^I_2 = \mathit{Win}^I(M_1, \Box\varphi^I_1)$ and $\varphi^O_2 = \mathit{Win}^I(M_1, \Box\varphi^O_1)$. Thus, in the following, unless differently specified, we only consider automata in normal form.

Definition 9 (Well-formed Automaton). We say that M is *well-formed* iff it is in normal form, and $\varphi^I_M \cap \varphi^O_M \neq \emptyset$.

Lemma 1. *If M is in normal form, then it holds:*

$$\forall s \in \varphi^I_M \,.\, \forall a \in \Gamma^O(M,s) \,.\, \widehat{\tau}^O_M(a,s) \subseteq \varphi^I_M$$
$$\forall s \in \varphi^O_M \,.\, \forall a \in \Gamma^I(M,s) \,.\, \widehat{\tau}^O_M(a,s) \subseteq \varphi^O_M.$$

Proof. For the first statement, by contradiction, suppose there is $s \in \varphi^I_M$ and $a \in \Gamma^O(M,s)$ such that $\widehat{\tau}^O_M(a,s) \not\subseteq \varphi^I_M$. Then $s \notin \mathit{Win}^I(M, \Box\varphi^I_M)$, because there is no way for the Input player to prevent output a to be carried out (see Definition 3). This contrasts with the assumption that M is in normal form. The second statement can be proven along similar lines.

2.3 Compatibility and Composition

In this subsection, we define the composition of two automata $M_1 = (Act_1, S_1, \tau_1^I, \tau_1^O, \varphi_1^I, \varphi_1^O)$ and $M_2 = (Act_2, S_2, \tau_2^I, \tau_2^O, \varphi_2^I, \varphi_2^O)$. We first define the product between $M_1 \otimes M_2$ as the classical automata-theoretic product, where M_1 and M_2 synchronize on shared actions and evolve independently on non-shared ones. We then identify a set of incompatible states where M_1 can do an output transition that is not accepted by M_2 or vice-versa. Finally, we obtain the composition $M_1 \| M_2$ from $M_1 \otimes M_2$ by strengthening the input assumptions of $M_1 \otimes M_2$ in such a way that M_1 and M_2 mutually satisfy their input assumptions.

Definition 10. We define the set of shared actions of M_1 and M_2 by:

$$Shared(M_1, M_2) = Act_1 \cap Act_2.$$

The product of two automata M_1 and M_2 is an automaton $M_1 \otimes M_2$, representing the joint behavior of M_1 and M_2. Similarly to other interface models, for each shared action, the output transitions of M_1 synchronize with the input transitions of M_2, and symmetrically, the output transitions of M_2 are synchronized with the input transitions of M_1. This models communication, and gives rise to output transitions in the product. The input transitions of M_1 and M_2 corresponding to shared actions are also synchronized, and lead to input transitions in the product. Output transitions, on the other hand, are not synchronized. If both M_1 and M_2 can emit a shared action a, they do so asynchronously, so that their output transitions interleave. As usual, the automata interleave asynchronously on transitions labeled by non-shared actions.

Definition 11 (Product). The *product* $M_1 \otimes M_2$ is the automaton $M_{12} = (Act_{12}, S_{12}, \tau_{12}^I, \tau_{12}^O, \varphi_{12}^I, \varphi_{12}^O)$, consisting of the following components.

- $Act_{12} = Act_1 \cup Act_2$; $\quad S_{12} = S_1 \times S_2$.
- $\varphi_{12}^I = \varphi_1^I \times \varphi_2^I$; $\quad \varphi_{12}^O = \varphi_1^O \times \varphi_2^O$.
- For $a \in Shared(M_1, M_2)$,

$$\langle s', t' \rangle \in \tau_{12}^O(a, \langle s, t \rangle) \text{ iff } \begin{cases} s' \in \tau_1^O(a, s) \text{ and } t' \in \tau_2^I(a, t) \text{ or} \\ t' \in \tau_2^O(a, t) \text{ and } s' \in \tau_1^I(a, s) \end{cases}$$

$$\langle s', t' \rangle \in \tau_{12}^I(a, \langle s, t \rangle) \text{ iff } s' \in \tau_1^I(a, s) \text{ and } t' \in \tau_2^I(a, t).$$

- For $a \in Act_1 \setminus Act_2$,

$$\langle s', t \rangle \in \tau_{12}^O(a, \langle s, t \rangle) \text{ iff } s' \in \tau_1^O(a, s)$$

$$\langle s', t \rangle \in \tau_{12}^I(a, \langle s, t \rangle) \text{ iff } s' \in \tau_1^I(a, s).$$

- For $a \in Act_2 \setminus Act_1$,

$$\langle s, t' \rangle \in \tau_{12}^O(a, \langle s, t \rangle) \text{ iff } t' \in \tau_2^O(a, t)$$

$$\langle s, t' \rangle \in \tau_{12}^I(a, \langle s, t \rangle) \text{ iff } t' \in \tau_2^I(a, t).$$

Example 3. The sociable interface automaton depicted in Figure 2(a) is the product $C \otimes D_1$ of the automata depicted in Figures 1(a) and 1(b). For instance, the input transition *fi re?* from state $\langle 1,1 \rangle$ to state $\langle 2,1 \rangle$ is obtained by combining the input transition *fi re?* from state 1 to state 2 in C with the input transition *fi re?* from state 1 to state 1 in D_1. The output transition *FD!* from state $\langle 1,2 \rangle$ to state $\langle 2,3 \rangle$ is obtained by combining the input transition *fi re?* from state 1 to state 2 in C with the output transition *fi re!* from state 2 to state 3 in D_1.

We have the following theorem.

Theorem 1. *The product is a commutative and associative operation, up to isomorphism.*

The product $M_{12} = M_1 \otimes M_2$ may contain states in which one of the components, say M_1, can do an output transition labeled by a shared action while the other component cannot do the corresponding input transition. This constitutes a violation of the input assumptions of M_2. We formalize such notion by introducing a *local compatibility* condition. To this end, for $p \in \{I, O\}$, we denote by $En^p(M,a)$ the set of states of M where the action a is enabled as input if $p = I$, and as output if $p = O$. Formally,

$$En^p(M,a) = \{s \in S_M \mid \widehat{\tau}^p_M(a,s) \neq \emptyset\}.$$

Definition 12 (Local Compatibility). Given $\langle s,t \rangle \in S_{12}$, $\langle s,t \rangle \in good(M_1,M_2)$ iff, for all $a \in Shared(M_1,M_2)$ the following conditions hold:

$$s \in En^O(M_1,a) \Rightarrow t \in En^I(M_2,a)$$
$$t \in En^O(M_2,a) \Rightarrow s \in En^I(M_1,a).$$

Example 4. Consider the product $C \otimes D_1$ of Example 4. The state $\langle 3,2 \rangle$ does not satisfy the Local Compatibility condition because, from state 2, D_1 can issue an output transition *fi re!*, and this cannot be matched by an input transition *fi re?* from state 3 of the control unit.

The composition of M_1 and M_2 is obtained from the product $M_1 \otimes M_2$ by strengthening the input assumptions of $M_1 \otimes M_2$ to avoid states that are not in $good(M_1,M_2)$. This is done by restricting the input invariant φ^I_{12} as shown in the next definition. The reason for restricting only the input behavior is that, when composing automata, only their input assumptions can be strengthened to ensure that no incompatibility arises, while their output behavior cannot be modified.

Definition 13 (Composition). Assume M_1 and M_2 are compatible. The *composition* $M_1 \| M_2$ is a sociable interface automaton identical to $M_1 \otimes M_2$, except that $\varphi^I_{M_1\|M_2} = \varphi^I_{12} \cap Win^I(M_{12}, \Box(\varphi^I_{12} \cap good(M_1,M_2)))$.

Definition 14 (Compatibility). We say that M_1 and M_2 are *compatible* if $\varphi^I_{M_1\|M_2} \cap \varphi^O_{M_1\|M_2} \neq \emptyset$.

The following theorem states that once the input transition relations have been strengthened, the automaton is in normal form: it is not necessary to also strengthen the output transition relations. This result thus provides a sanity check, since strengthening the output transitions means restricting the output behavior of the interfaces, which is not reasonable.

Theorem 2. *If M_1 and M_2 are compatible, and they are in normal form, then $M_1 \| M_2$ is in normal form.*

The following result implies that the automata can be composed in any order.

Theorem 3. *The composition is a commutative and associative operation, up to isomorphism.*

3 Sociable Interfaces with Variables

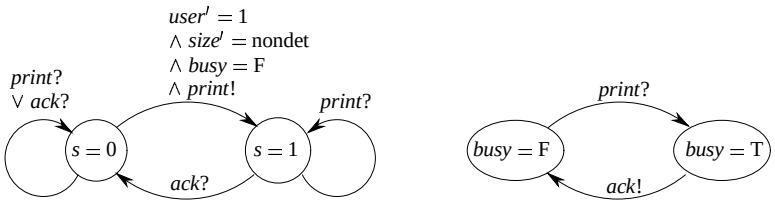
3.1 Preview

In modeling systems and designs, it is often valuable to have a notion of global state, which can be read and updated by the various components of the system. A common, and flexible, paradigm consists in having the global state consist of a value assignment to a set of global variables. Once the global state is represented by global variables, it is natural to encode also the local state of each component via (local) variables.

Previous interface models, such as interface automata [6, 8] and interface modules [7, 3] were based on either actions, or variables, but not both. In sociable interfaces, however, we want to have both: actions to model synchronization, and variables to encode the global and local state of components. In this, sociable interfaces are closely related to the *I/O Automata Language* (IOA) of [11].

Interface models are games between Input and Output, and in the models, it is essential that Input and Output are able to choose their moves independently from one another. To this end, in previous interface formalisms with variables, the variables were partitioned into *input* and *output* variables [7, 3]. A move of Input consisted in choosing the next value of the input variables, and a move of Output consisted in choosing the next value of the output variables: this ensured the independence of the moves. Consequently, interfaces sharing output variables could not be composed, and in a composite system, every variable that was not input from the environment was essentially "owned" by one of the component interfaces, which was the only one allowed to modify its value.

In sociable interface modules, we can leverage the presence of actions in order to achieve a more general setting, in which variables can be modified by more than one module. Informally, the model is as follows. With each action, we associate a set of variables that can be modified by the action, as well as an output and an input transition relation that describe the ways in which the variables can be modified when the component, or its environment, output the action. When the Output player takes an action a, the output transition relation associated with a specifies how the player can update the variables associated with a. Symmetrically, when the Input player takes an action a, the input transition relation associated with a specifies what changes to the variables associated with a can be accepted by the module.



(a) U_1: User 1 (b) P: Printer.

Fig. 5. Informal depiction of the user process, printer, and printer adapter interfaces in the setting with variables.

When modules are composed, actions synchronize in the same way as they do in sociable interface automata. When an output event $a!$ of module M synchronizes with an input event $a?$ of module N, we must check that all variable updates that can accompany $a!$ from M are acceptable to N, that is, that the output transition relation associated with a in M respects the constraints specified by the input transition relation associated with a in N. Empty transition relations are used to rule out the possibility of taking an action as output or input.

3.2 An Example: Modeling a Print Server

We illustrate the main features of sociable interface modules through a very simple example: a model of a shared print server. The model consists of modules representing the print server, as well as user processes that communicate with the server to print jobs. The modules composing this example are depicted in an intuitive fashion in Figure 5; the actual input to the tool TIC for this model is given in Figure 6, and it will be described later.

The user module U_1 (Figure 5(a)) communicates via two actions: an action *print*, whose output represents a print request, and an action *ack*, whose input represents an acknowledgment. When generating *print* as an output, U_1 updates the global variables *user* and *size*, which indicate the user who issued the request, and the size of the request. The print server P (Figure 5(b)) synchronizes on *ack* and *print*, and also updates a global state variable *busy*, indicating whether the printer is busy. To ensure compatibility, the user module checks that $busy = F$ before printing. In addition, to ensure compatibility in presence of multiple user modules, the user module ignores inputs *ack* when idle ($s = 0$), as these acknowledgments are directed to other users, and ignores all inputs *print*, as these correspond to input requests from other users.

3.3 Definitions

We assume a fixed set $\mathcal{V}$ of variables. All variables in $\mathcal{V}$ are interpreted over a given domain $\mathcal{D}$. Given $V \subseteq \mathcal{V}$, a *state* over V is a mapping $s : V \to \mathcal{D}$ that associates with

each $x \in V$ a value $s(x) \in \mathscr{D}$. For a set of variables $U \subseteq V$, and a state $s \in [\![V]\!]$, the restriction of s to U is a state $t \in [\![U]\!]$ denoted as $s[U]$. For two disjoint sets of variables V_1 and V_2, and two states $s_1 \in [\![V_1]\!]$ and $s_2 \in [\![V_2]\!]$, the operation $(s_1 \circ s_2)$ composes the two states resulting in a new state $s = s_1 \circ s_2 \in [\![V_1 \cup V_2]\!]$, such that $s(x) = s_1(x)$ for all $x \in V_1$ and $s(x) = s_2(x)$ for all $x \in V_2$.

Our formal model with variables is called a *sociable interface module*. It is convenient to define sociable interface modules with respect to a predicate representation. Given a set V of variables, we denote by $Preds(V)$ the set of first-order predicate formulas with free variables in V; we assume that these predicates are written in some specified first-order language with interpreted function symbols and predicates; in our tool, the language contains some arithmetic operators, relational symbols, and boolean connectives. Given a set of variables V, we let $V' = \{x' \mid x \in V\}$ be the set consisting of primed versions of variables in V. A variable $x' \in V'$ represents the *next value* of $x \in V$. Given a formula $\psi \in Preds(V)$ and a state $s \in [\![V]\!]$, we write $s \models \psi$ if the predicate formula ψ is true when its free variables are interpreted as specified by s. Given a formula $\rho \in Preds(V \cup V')$ and two states $s, s' \in [\![V]\!]$, we write $\langle s, s' \rangle \models \rho$ if the formula ρ holds when its free variables $x \in V$ are interpreted as $s(x)$, and its free variables $x' \in V'$ are interpreted as $s'(x)$. Given a set U of variables, we define the formula:

$$Unchgd(U) \;=\; \bigwedge_{x \in U} (x' = x),$$

which states that the variables in U do not change their value in a transition. Given a predicate $\psi \in Preds(V)$, we denote by ψ' the predicate obtained by substituting x with x' in ψ, for all $x \in V$.

With these definitions, we can define sociable interface modules as follows.

Definition 15 (Sociable Interface Module). A *sociable interface module* (*module*, for short) is a tuple $M = (Act, V^G, V^L, V^H, W, \rho^{IL}, \rho^{IG}, \rho^O, \psi^I, \psi^O)$, where:

- Act is a set of *actions*.
- V^G is a set of *global variables*, V^L is a set of *local variables*, and $V^H \subseteq V^G$ is a set of *history variables*. We require $V^L \cap V^G = \emptyset$. We set $V^{\text{all}} = V^L \cup V^G$ and $V = V^L \cup V^H$.
- $W : Act \rightrightarrows V^{\text{all}}$ associates with each $a \in Act$ the set of variables $W(a) \subseteq V^{\text{all}}$ that can be modified by a.
- For each $a \in Act$, the predicate $\rho^{IL}(a) \in Preds(V^{\text{all}} \cup (V^{\text{all}})')$ is the *input local transition predicate* for a. We require this transition predicate to be *deterministic* w.r.t. variables in V^L, that is, for all $a \in Act$, all $s \in [\![V^{\text{all}}]\!]$, and all $t \in [\![(V^G)']\!]$, there is a unique $u \in [\![(V^L)']\!]$ such that $s \circ t \circ u \models \rho^{IL}(a)$.
- For each $a \in Act$, the predicate $\rho^{IG}(a) \in Preds(V^{\text{all}} \cup (V^G)')$ is the *input global transition predicate* for a.
- For each $a \in Act$, the predicate $\rho^O(a) \in Preds(V^{\text{all}} \cup W(a)')$ is the *output transition predicate* for a.
- $\psi^I \in Preds(V^{\text{all}})$ is the *input invariant predicate*.
- $\psi^O \in Preds(V^{\text{all}})$ is the *output invariant predicate*.

A *state* is a value assignment to V^{all}; we denote the set of states of the module by $S = [\![V^{\mathrm{all}}]\!]$. The invariant predicates define invariants

$$\varphi^I = \{s \in S \mid s \models \psi^I\}, \qquad \varphi^O = \{s \in S \mid s \models \psi^O\}.$$

As a shorthand, for all $a \in Act$ we let $\rho^I(a) = \rho^{IL}(a) \wedge \rho^{IG}(a)$, and we define

$$\widehat{\rho}^I(a) = \rho^I(a) \wedge (\psi^I)'$$
$$\widehat{\rho}^O(a) = \rho^O(a) \wedge (\psi^O)' \wedge \mathit{Unchgd}(V^{\mathrm{all}} \setminus W(a)).$$

Notice that $\widehat{\rho}^I(a)$ and $\widehat{\rho}^O(a)$ are predicates over $V^{\mathrm{all}} \cup (V^{\mathrm{all}})'$.

In our model, each module owns a set of local variables, that describe the internal state of a component. We distinguish a set V^H of *history* variables, and a set $V^G \setminus V^H$ of *history-free* variables. A module must be aware of all actions that can modify its history variables (see, in the following, the *non-interference* condition in Definition 19). On the other hand, history-free variables can be modified by environment actions that are not known to the module. The distinction between the history and history-free global variables is thus used to limit the amount of actions a module should include; this point will be clarified when we will discuss module composability.

The definitions of the input and output transition relations are similar to those of Section 2. We require the input transition relation to be deterministic on local variables. This assumption corresponds to the assumption, in the model without variables, that input transitions are deterministic. In fact, we will see that when an output and an input transitions synchronize, it is the output transition that selects the next value of the global variables, and the input transition is used only to determine the next value of the local variables.

In the remainder of this section we consider a fixed module $M = (Act_M, V_M^G, V_M^L, V_M^H, W_M, \rho_M^{IL}, \rho_M^{IG}, \rho_M^O, \psi_M^I, \psi_M^O)$, and we set $V_M = V_M^L \cup V_M^H$, $V_M^{\mathrm{all}} = V_M^L \cup V_M^G$, and correspondingly for the shorthands $\widehat{\rho}_M^I$ and $\widehat{\rho}_M^O$.

Definition 16 (Set of States). The set of states of the sociable interface module M is given by $S_M = [\![V_M^{\mathrm{all}}]\!]$.

The sets of moves for players Input and Output are defined as follows. Note that, when Input plays the move Δ_0, Input can also choose a new assignment to the history-free variables. This models the fact that history-free variables can be modified by environment actions that are not known to the module.

Definition 17 (Moves). The sets $\Gamma^I(M,s)$ and $\Gamma^O(M,s)$ of Input and Output moves at $s \in S_M$ are defined as follows:

$$\begin{aligned}\Gamma^I(M,s) = &\{\Delta_0\} \times \{s' \in [\![V_M^{\mathrm{all}}]\!] \mid s'[V_M] = s[V_M]\} \cup \\ &\{\langle a, s'\rangle \in Act_M \times [\![V_M^{\mathrm{all}}]\!] \mid \langle s, s'\rangle \models \widehat{\rho}_M^I(a)\}\end{aligned}$$
$$\Gamma^O(M,s) = \{\Delta_0\} \cup \{\langle a, s'\rangle \in Act_M \times [\![V_M^{\mathrm{all}}]\!] \mid \langle s, s'\rangle \models \widehat{\rho}_M^O(a)\}.$$

The outcome of the moves are as follows.

Definition 18 (Move Outcome). For all states $s \in S_M$ and moves $m^I \in \Gamma^I(M,s)$ and $m^O \in \Gamma^O(M,s)$, the *outcome* $\delta(M,s,m^I,m^O) \subseteq S_M$ of playing m^I and m^O at s can be defined as follows.

$$\delta(M,s,\langle\Delta_0,s'\rangle,\Delta_0) = \{s'\}, \qquad \delta(M,s,\langle\Delta_0,s'\rangle,\langle a,t'\rangle) = \{s',t'\},$$
$$\delta(M,s,\langle a,s'\rangle,\Delta_0) = \{s'\}, \qquad \delta(M,s,\langle a,s'\rangle,\langle b,t'\rangle) = \{s',t'\}.$$

The definitions of run, strategy, strategy outcome, winning state and well-formedness are similar to the ones given in Section 2.

3.4 The Printer Example, Continued

Figure 6 presents our print-server example, encoded in the actual input language of the tool TIC. The system consists of the global variables *busy*, *size*, *user*, of a printer module, and of two user modules. In each module, we give the set of history-free variables (called *stateless* in the language of the tool); the set of global variables of the module is simply inferred as the set of global variables that appear anywhere in the module.

The module `Printer` communicates via two actions, *ack* and *print*. The transition predicates of these actions are specified using a guarded-commands syntax, similar to [4, 1]. Each guarded command has the form *guard* $\Rightarrow$ *command*, where *guard* and *command* are formulas written over the set of primed and unprimed variables. A guarded command *guard* $\Rightarrow$ *command* can be taken when its guard is true; when taken, *command* specify how the variables are updated. For instance, the output transition *print* in module `User1` can be taken when $s = 0$ and $\neg$*busy*, and it leads to a state where $s = 1$ and *user* $= 1$. The value of *size* in the destination state is nondeterministic.

When specifying sociable interface modules in the tool TIC, we use several shorthands to make the notation more pleasant:

- When we do not specify the input or output transition relation for an action, the omitted transition relations are assumed to be false. For example, the action *ack* has no input transition relation in the printer: this specifies that no other module should be able to emit it. Similarly, the action *ack* has no output transition relation in the user modules, specifying that modules do not generate it.
- When we specify a transition relation via an empty guarded command, the guard is assumed to be always true, and the command is as follows:
  - *Output transition relations, and local part of input transitions:* no variables are changed.
  - *Global part of input transitions:* the transition relation is considered to be *true*, so that all state changes are accepted.
- In a guarded command *guard* $\Rightarrow$ *command*, when *guard* is missing, it is assumed to be true. If *command* is missing, then:
  - *Output transitions, and local part of input transitions:* no variables are changed.
  - *Global part of input transitions:* the transition relation is considered to be *true*, so that all state changes are accepted.

```
var busy: bool;    // global variable indicating a printer busy
var size: [0..10]; // size of the print job
var user: [0..5];  // user who requested the job

module Printer:

  output ack   { busy ==> not busy'; }
                                // ack? is not allowed

  input  print { global: not busy ==> busy'; }

endmodule


module User1:
  var s: [0..1];
  stateless size, user;

  output print { s = 0 & not busy ==>
                                  s' = 1 & user' = 1 & nondet size'; }

  input  print { }  // print? is allowed and ignored

  input  ack   { local: s = 1 ==> s' := 0;
                 else  s = 0 ==> ;        }  // ignore ack? when s=0

endmodule


module User2:
  var s: [0..1];
  stateless size, user;

  output print { s = 0 & not busy ==>
                                  s' = 1 & user' = 2 & nondet size'; }

  input  print { }  // print? is allowed and ignored

  input  ack   { local: s = 1 ==> s' := 0;
                 else  s = 0 ==> ;        }  // ignore ack? when s=0

endmodule
```

Fig. 6. TIC input modeling a simple print server.

```
var busy: bool;    // global variable indicating a printer busy
var size: [0..10]; // size of the print job
var user: [0..5];  // user who requested the job

module Printer:
  output ack   { busy & size < 5 ==> not busy'; } // accept if size < 5
                                // ack? is not allowed

  output nack  { busy & size > 4 ==> not busy'; } // reject if size < 5
                                // nack? is not allowed

  input  print { global: not busy ==> busy'; }
endmodule

module User1:
  var s: [0..1];
  stateless size, user;

  output print { s = 0 & not busy ==>
                                  s' = 1 & user' = 1 & nondet size'; }
  input  print { }  // print? is allowed and ignored

  input  ack   { local: s = 1 ==> s' := 0;
                 else  s = 0 ==> ;        }  // ignore ack? when s=0

  input  nack  { local: s = 1 ==> s' := 0;
                 else  s = 0 ==> ;        }  // ignore nack? when s=0
  endmodule

module User2:
  var s: [0..1];
  stateless size, user;

  output print { s = 0 & not busy ==>
                                  s' = 1 & user' = 2 & nondet size'; }
  input  print { }  // print? is allowed and ignored

  input  ack   { local: s = 1 ==> s' := 0;
                 else   s = 0 ==> ;        }  // ignore ack? when s=0

  input  nack  { local: s = 1 ==> s' := 0;
                 else   s = 0 ==> ;        }  // ignore nack? when s=0
endmodule
```

Fig. 7. TIC input modeling a print server that rejects large jobs.

– In output transitions, and in the local part of input transitions, variables that are not mentioned primed in the *command* portion of a guarded command *guard* $\Rightarrow$ *command* do not change their value.

As a more elaborate example, in Figure 7 we present the code of a print server that can accept or reject jobs, depending on their length.

3.5 Compatibility and Composition

We now describe the composition of two modules. Due to the presence of variables, this process is more involved than the one presented in Section 2.

The composition of two modules M_1 and M_2 is defined in four steps, in a similar way as stated in [9]. First, we define when M_1 and M_2 are *composable*, and in the affirmative case, we define their *product* $M_1 \otimes M_2$. On the resulting product module, we identify a set of *bad states:* these are the states where M_1 (resp. M_2) can produce an output that is not accepted by M_2 (resp. M_1). Finally, the *composition* $M_1 \| M_2$ of M_1 and M_2 is obtained from the product $M_1 \otimes M_2$ by strengthening the input transition relations of $M_1 \otimes M_2$ in such a way that all bad states are avoided.

In the following, we consider two modules M_1 and M_2, where $M_i = (Act_i, V_i^G, V_i^L, V_i^H, W_i, \rho_i^{IL}, \rho_i^{IG}, \rho_i^O, \psi_i^I, \psi_i^O)$, for $i = 1, 2$, and we let $V_i = V_i^L \cup V_i^H$ and $V_i^{\text{all}} = V_i^L \cup V_i^G$.

We say that two modules M_1 and M_2 are *composable* if they have disjoint sets of local variables, and if they satisfy a *non-interference* condition, stating that if an action of a module can modify a state variable of the other, then the action is shared. This condition ensures that the set of actions of a module includes all the actions that can modify its state variables. This condition is essential for modular reasoning. It ensures that composition does not add behaviors: all changes in the state of m_1 caused by modules with which M_1 is composable can be already explained by the input transitions associated with actions of M_1.

Definition 19 (Composability). Two sociable interface modules M_1 and M_2 are *composable* iff $V_1^L \cap V_2^L = \emptyset$ and if the following *non-interference* conditions hold:

$$\forall a \in Act_2 \,.\, W_2(a) \cap V_1 \neq \emptyset \implies a \in Act_1$$
$$\forall a \in Act_1 \,.\, W_1(a) \cap V_2 \neq \emptyset \implies a \in Act_2.$$

The non-interference condition is the main justification for distinguishing between the sets of history and history-free variables. The non-interference condition states that a module should know all actions of other modules that modify its history variables. If we dropped the distinction, requiring that a module knows all actions of other modules that can change any of its variables (history or history-free), we could greatly increase the number of actions that must be known to the module.

As an example, consider a set of modules $\{N_i\}_{i \in \{1..100\}}$. Each module has an action a_i whose output transition relation sets *index* to i, and x to some content, where *index* and x are global variables shared among all $N_1, \ldots, N_{100}$. If module N_i does not need to keep track of the value of *index* and x, as these variables are used as outputs only, then we can

let $index \notin V_{N_i}$ and $x \notin V_{N_i}$, even though of course $index, x \in V_{N_i}^{\text{all}}$. The non-interference condition for N_i, stated in terms of V_{N_i}, will not require N_i to know about a_j for $i \neq j$. This keeps the model of N_i simple and concise and, even more importantly, enables us to model N_i before we know exactly how many other modules there are that can modify *index* and x. Dropping the distinction between V_{N_i} and $V_{N_i}^{\text{all}}$, on the other hand, would force each N_i to have all the actions $a_1, \ldots, a_{100}$ in its set of actions, greatly complicating the model, and forcing us to know in advance how many components there are, before each of the components can be modeled. Similarly, if a module reads a variable x, but does not need to know how and when the value of x is changed, then the variable x can be declared to be history-free, so that the module does not have to know all the actions that can modify x. Hence, the distinction between history and history-free variables is at the heart of our "sociable" approach to compositional modeling.

We define the product of two sociable interface modules M_1 and M_2 as follows.

Definition 20 (Product). Assume that M_1 and M_2 are composable. The *product* $M_1 \otimes M_2$ is the interface $M_{12} = (Act_{12}, V_{12}^G, V_{12}^L, V_{12}^H, W_{12}, \rho_{12}^{IL}, \rho_{12}^{IG}, \rho_{12}^O, \psi_{12}^I, \psi_{12}^O)$, defined as follows.

- $Act_{12} = Act_1 \cup Act_2$.
- $V_{12}^G = V_1^G \cup V_2^G; \quad V_{12}^L = V_1^L \cup V_2^L; \quad V_{12}^H = V_1^H \cup V_2^H$.
- $W_{12}(a) = \begin{cases} W_1(a) \cup W_2(a) \cup V_1^L \cup V_2^L & \text{for } a \in Shared(M_1, M_2) \\ W_i(a) & \text{for } a \in Act_i \setminus Act_{3-1}, i \in \{1,2\}. \end{cases}$
- $\psi_{12}^I = \psi_1^I \wedge \psi_2^I; \quad \psi_{12}^O = \psi_1^O \wedge \psi_2^O$.
- For $a \in Shared(M_1, M_2)$, we let:

$$\rho_{12}^O(a) = \\ = \begin{pmatrix} \exists (V_{12}^G)' \setminus W_{12}(a)' \,.\, \rho_1^O(a) \wedge \rho_2^{IL}(a) \wedge \rho_2^{IG}(a) \wedge Unchgd(V_{12}^{\text{all}} \setminus (W_1(a) \cup V_2^L)) \\ \vee \\ \exists (V_{12}^G)' \setminus W_{12}(a)' \,.\, \rho_2^O(a) \wedge \rho_1^{IL}(a) \wedge \rho_1^{IG}(a) \wedge Unchgd(V_{12}^{\text{all}} \setminus (W_2(a) \cup V_1^L)) \end{pmatrix}$$

$$\rho_{12}^{IL}(a) = \rho_1^{IL}(a) \wedge \rho_2^{IL}(a)$$

$$\rho_{12}^{IG}(a) = \rho_1^{IG}(a) \wedge \rho_2^{IG}(a).$$

- For $i \in \{1,2\}$ and $a \in Act_i \setminus Act_{3-i}$ we let:

$$\rho_{12}^O(a) = \rho_i^O(a)$$

$$\rho_{12}^{IL}(a) = \rho_i^{IL}(a) \wedge Unchgd(V_{3-i}^L)$$

$$\rho_{12}^{IG}(a) = \rho_i^{IG}(a) \wedge Unchgd(V_{3-i}^H).$$

We have the following result.

Theorem 4. *Product between modules is a commutative and associative operation.*

Similarly to Definition 12, we identify a set of locally incompatible states of the product $M_1 \otimes M_2$.

Definition 21 (Local Compatibility). Given $s \in [\![V_{12}^{\mathrm{all}}]\!]$, we say that s is *good* iff it satisfies the predicate $good(M_1, M_2)$, defined as follows:

$$good(M_1, M_2) = \\ = \bigwedge_{a \in Shared(M_1, M_2)} \begin{pmatrix} \forall (V_{12}^{\mathrm{all}})' . \left(\left(\widehat{\rho}_1^O(a) \wedge Unchgd(V_2^G \setminus W_1(a)) \right) \implies \widehat{\rho}_2^{IG}(a) \right) \\ \wedge \\ \forall (V_{12}^{\mathrm{all}})' . \left(\left(\widehat{\rho}_2^O(a) \wedge Unchgd(V_1^G \setminus W_2(a)) \right) \implies \widehat{\rho}_1^{IG}(a) \right) \end{pmatrix}.$$

Using this condition, the composition $M_1 \| M_2$ is obtained from $M_1 \otimes M_2$ by restricting the input invariant of M_{12} to the set of well-formed states from where input has a strategy to always stay in the good states $good(M_1, M_2)$, in analogy with Definition 13.

Theorem 5. *Composition between modules is a commutative and associative operation.*

4 Refinement

We wish to define a refinement relation between modules, such that when M_1 refines M_2, M_1 can be used as a replacement for M_2 in any context. First, some conditions should hold on the set of variables that the modules manipulate. In the following, M_1 and M_2 are two modules in normal form. For $i \in \{1,2\}$, let $M_i = (Act_i, V_i^G, V_i^L, V_i^H, W_i, \rho_i^{IL}, \rho_i^{IG}, \rho_i^O, \psi_i^I, \psi_i^O)$, $V_i = V_i^H \cup V_i^L$ and $S_i = [\![V_i]\!]$. The sets Act_i, V_i^G, V_i^H, and W_i jointly define the *signature* of a module M_i.

Definition 22 (Signature). The signature $Sign(M_i)$ of a module $M_i = (Act_i, V_i^G, V_i^L, V_i^H, W_i, \rho_i^{IL}, \rho_i^{IG}, \rho_i^O, \psi_i^I, \psi_i^O)$, is the tuple $(Act_i, V_i^G, V_i^H, W_i)$.

The following result shows that signature equality preserves composability. It can be proved by inspecting Definition 19.

Theorem 6. *Let N_1, N_2, and N_3 be three modules, such that the $Sign(N_1) = Sign(N_2)$, and N_2 and N_3 are composable. For $i \in \{1,2,3\}$, let V_i^L be the set of local variables of N_i. If $V_1^L \cap V_3^L = \emptyset$, then N_1 and N_3 are composable.*

To replace M_2, M_1 should also behave like it, from the point of view of the environment. As usual in a game-theoretic setting such as ours, this constraint is captured by *alternating simulation* [2]. Intuitively, M_1 must be willing to accept at least all the inputs that M_2 accepts, and it should emit a subset of the outputs emitted by M_2.

Definition 23 (Alternating Simulation). Assume that $Sign(M_1) = Sign(M_2)$. A relation $\preceq \subseteq S_1 \times S_2$ is an *alternating simulation* iff $s \preceq t$ implies:

1. $s[V_1^G] = t[V_1^G]$;
2. for all $a \in Act_1$ and for all $t' \in S_2$ such that $\langle t, t' \rangle \models \widehat{\rho}_2^I(a)$ there exists $s' \in S_1$ such that $\langle s, s' \rangle \models \widehat{\rho}_1^I(a)$ and $s' \preceq t'$;
3. for all $a \in Act_1$ and for all $s' \in S_1$ such that $\langle s, s' \rangle \models \widehat{\rho}_1^O(a)$ there exists $t' \in S_2$ such that $\langle t, t' \rangle \models \widehat{\rho}_2^O(a)$ and $s' \preceq t'$.

We say that s is similar to t, and we write $s \sqsubseteq t$, if there exists an alternating simulation $\preceq$ such that $s \preceq t$. Similarity is itself a simulation (the coarsest one). For M_1 to refine M_2, M_1 and M_2 should have the same signature, and each well-formed state of M_2 must be similar to some well-formed state of M_1.

Definition 24 (Refinement). We say that M_1 *refines* M_2 iff *(i)* $Sign(M_1) = Sign(M_2)$, and *(ii)* for all $t \models \psi_2^I \wedge \psi_2^O$ there is $s \models \psi_1^I \wedge \psi_1^O$ such that $s \sqsubseteq t$.

Theorem 7. *Let N_1, N_2, and N_3 be three modules, such that N_1 refines N_2, and N_2 and N_3 are compatible. For $i \in \{1,2,3\}$, let V_i^L be the set of local variables of N_i. If $V_1^L \cap V_3^L = \emptyset$, then N_1 and N_3 are compatible.*

We now introduce the related concept of *bisimilarity*. Bisimilarity between two modules captures the intuitive concept that the environment cannot distinguish the two modules.

Definition 25 (Alternating Bisimulation). Assume that $Sign(M_1) = Sign(M_2)$. A relation $\approx \subseteq S_1 \times S_2$ is an *alternating bisimulation* iff it is a *symmetrical* alternating simulation.

We say that s and t are *bisimilar*, and we write $s \cong t$, if there exists an alternating bisimulation $\approx$ such that $s \approx t$.

Definition 26 (Bisimilarity). We say that M_1 and M_2 are *bisimilar* iff *(i)* $Sign(M_1) = Sign(M_2)$, and *(ii)* for all $t \models \psi_2^I \wedge \psi_2^O$ there is $s \models \psi_1^I \wedge \psi_1^O$ such that $s \cong t$, and for all $s \models \psi_1^I \wedge \psi_1^O$ there is $t \models \psi_2^I \wedge \psi_2^O$ such that $s \cong t$.

Theorem 8. *Let N_1, N_2, and N_3 be three modules, such that N_1 is bisimilar to N_2. For $i \in \{1,2,3\}$, let V_i^L be the set of local variables of N_i. If $V_1^L \cap V_3^L = \emptyset$ and $V_2^L \cap V_3^L = \emptyset$, then N_1 and N_3 are compatible iff N_2 and N_3 are compatible.*

5 Symbolic Implementation

In this section, we examine the problem of efficiently implementing the following operations: *(i)* module composition, *(ii)* verification of safety properties of modules (such as well-formedness), and *(iii)* refinement and bisimilarity checking between modules.

Consider the module $M = (Act_M, V_M^G, V_M^L, V_M^H, W_M, \rho_M^{IL}, \rho_M^{IG}, \rho_M^O, \psi_M^I, \psi_M^O)$, and set $V_M^{\text{all}} = V_M^L \cup V_M^G$.

A well-established technique for efficiently implementing finite transition systems is based on MDDs [12, 14]. MDDs are graph-like data structures that allow us to represent and manipulate functions of the type $A \to \{\mathrm{T}, \mathrm{F}\}$, for a finite set A (i.e. predicates over A). Therefore, we assume that the variable domain $\mathscr{D}$ is finite, and we represent the predicates ρ_M^{IL}, ρ_M^{IG}, ρ_M^O, ψ_M^I, and ψ_M^O as MDDs. We now show that all the operations involved in computing the composition of modules, checking their well-formedness, checking safety properties, and checking refinement are computable on MDDs.

5.1 Safety Games

A basic operation on modules is computing the set of winning states for a player $p \in \{I, O\}$ w.r.t. a safety goal, that is $Win^p(M, \Box\varphi)$, for some set $\varphi \subseteq [\![V_M^{\text{all}}]\!]$. The operations of checking well-formedness, putting a module in normal form, and computing the composition of two modules, are all reducible to solving safety games.

By abuse of notation, we denote by $Win^p(M, \Box\varphi)$ both the set of states it denotes, and its characteristic function, which is a predicate over V_M^{all}.

It is well known that such set of winning states can be characterized as a fixpoint of an equation involving the so-called *controllable predecessors operators*. For a player $p \in \{I, O\}$ and a predicate $X \in Preds(V_M^{\text{all}})$, the operator $Cpre^p(X)$ returns the set of states from which player p can force the game into X in one step, regardless of the opponent's moves. Formally, we have the following definition.

Definition 27 (Controllable Predecessor Operator). For a predicate $X \in Preds(V_M^{\text{all}})$, we have:

$$Cpre^I(X) = \exists m^I \in \Gamma^I(M,s) \,.\, \forall m^O \in \Gamma^O(M,s) \,.\, \forall t \in \delta(M,s,m^I,m^O) \,.\, t \models X$$
$$Cpre^O(X) = \exists m^O \in \Gamma^O(M,s) \,.\, \forall m^I \in \Gamma^I(M,s) \,.\, \forall t \in \delta(M,s,m^I,m^O) \,.\, t \models X.$$

Intuitively, $Cpre^I(X)$ (resp.$Cpre^O(X)$) holds true for the states from which the Input (resp. Output) player has a move that leads to X for each possible counter-move of the Output (resp. Input) player. For all $\varphi \in Preds(V_M^{\text{all}})$, we have:

$$Win^I(M, \Box\varphi) = \nu X. \, [\varphi \wedge Cpre^I(X)]$$
$$Win^O(M, \Box\varphi) = \nu X. \, [\varphi \wedge Cpre^O(X)],$$

where $\nu X \,.\, f(X)$ denotes the greatest fixpoint of the operator f. Since $Cpre^I(\cdot)$ is monotonic, the above fixpoints exist and can be computed by Picard iteration:

$$X_0 = \varphi, \qquad X_{i+1} = \varphi \wedge Cpre^I(X_i), \qquad \ldots \qquad X_n = X_{n+1} = Win^I(M, \Box\varphi). \tag{1}$$

We now show how to compute $Cpre^I(X)$ starting from the MDD representation of M. Considering Definition 18, in order for a state s to satisfy $Cpre^I(X)$, two conditions must hold. First, every output transition should lead to X. Second, either $s \models X$, in which case Input can play $\langle \Delta_0, s \rangle$, or there must be an input transition that leads to X. This observation allows us to express $Cpre^I(X)$ as follows:

$$Cpre^I(X) = \forall Pre^O(X) \wedge \exists Pre^I(X),$$

where

$$\forall Pre^O(X) = \bigwedge_{a \in Act_M} \forall (V_M^{\text{all}})' \,.\, (\widehat{\rho}_M^O(a) \Rightarrow X')$$
$$\exists Pre^I(X) = X \vee \left(\exists (V_M^{\text{all}})' \,.\, X' \wedge Unchgd(V_M^H \cup V_M^L)\right) \vee \bigvee_{a \in Act_M} \exists (V_M^{\text{all}})' \,.\, (\widehat{\rho}_M^I(a) \wedge X').$$

Since boolean operations and quantifications of variables are computable on MDDs, the operators above are computable. In a dual fashion, $Cpre^O(X)$ can be computed from the non-game operators $\forall Pre^I(\cdot)$ and $\exists Pre^O(\cdot)$.

We can improve the efficiency of computing $Win^I(M, \Box\varphi)$, by observing that, since (1) is a decreasing sequence, it holds that $\nu X.\,[\varphi \wedge Cpre^I(X)] = \nu X.\,[\varphi \wedge X \wedge Cpre^I(X)]$. Since $X \wedge Cpre^I(X) = X \wedge \forall Pre^O(X)$, we obtain

$$Win^I(M, \Box\varphi) = \nu X.\,[\varphi \wedge X \wedge \forall Pre^O(X)] = \nu X.\,[\varphi \wedge \forall Pre^O(X)].$$

In conclusion, we can then compute $Win^I(M, \Box\varphi)$ by iterating $\forall Pre^O(\cdot)$ instead of the more complicated $Cpre^I(\cdot)$. A similar argument holds for the computation of $Win^O(M, \Box\varphi)$.

5.2 Composition

By inspecting Definition 20, it is clear that computing the product of two modules M_1 and M_2 only involves simple boolean operations on the predicates that define the modules. Such operations are computable on MDDs.

To obtain the composition $M_1 \| M_2$, according to Definition 13, the input invariant ψ^I_{12} of the product must be conjoined with the predicate $Win^I(M_1 \otimes M_2, \Box(\psi^I_{12} \wedge good(M_1, M_2))$. To compute the above winning set, we first compute the predicate $good(M_1, M_2)$ following Definition 21, and then solve the safety game as explained in Section 5.1.

5.3 Refinement

Let M_1 and M_2 be two modules in normal form, such that $Sign(M_1) = Sign(M_2)$. For $i \in \{1, 2\}$, let $M_i = (Act, V^G, V^L_i, V^H, W, \rho^{IL}_i, \rho^{IG}_i, \rho^O_i, \psi^I_i, \psi^O_i)$, $V^{\text{all}}_i = V^G \cup V^L_i$ and $S_i = [\![V^{\text{all}}_i]\!]$. Assume for simplicity that $V^L_1 \cap V^L_2 = \emptyset$. We wish to compute the coarsest alternating simulation $\sqsubseteq$ between S_1 and S_2. Consider the predicate $\psi_\sqsubseteq$ over the set of variables $V^{\text{all}}_1 \cup V^{\text{all}}_2$, defined as the greatest fixpoint of the operator $SimPre(\cdot)$, defined as follows. For all $X \in Preds(V^{\text{all}}_1 \cup V^{\text{all}}_2)$, we have

$$\begin{aligned} SimPre(X) = X \wedge & \bigwedge_{a \in Act} \forall (V^{\text{all}}_2)' . \exists (V^L_1)' . \left(\widehat{\rho}^I_2(a) \implies \widehat{\rho}^I_1(a) \wedge X'\right) \\ \wedge & \bigwedge_{a \in Act} \forall (V^{\text{all}}_1)' . \exists (V^L_2)' . \left(\widehat{\rho}^O_1(a) \implies \widehat{\rho}^O_2(a) \wedge X'\right). \end{aligned}$$

The operator $SimPre(\cdot)$, and consequently its fixpoint $\psi_\sqsubseteq$, can be computed from the MDD representation of M_1 and M_2. The following result states that $\psi_\sqsubseteq$ can be used to trivially obtain $\sqsubseteq$. The result can be proven by induction, observing that $SimPre(\cdot)$ represents conditions 2 and 3 of Definition 23.

Theorem 9. *Given $s \in S_1$ and $t \in S_2$, $s \sqsubseteq t$ iff $s[V^G] = t[V^G]$ and $s \circ t[V^L_2] \models \psi_\sqsubseteq$.*

A similar algorithm can be used to compute the coarsest bisimulation $\cong$.

6 Comparison with Previous Interface Models

The sociable interface model presented in this paper is closely related to the *I/O Automata Model* (IOA) of [11]: sociable interfaces synchronize on actions and use variables to encode the state of components. However, sociable interfaces diverge from *I/O Automata* in several ways. Unlike *I/O Automata*, where every state must be receptive to every possible input event, sociable interfaces allow states to forbid some input events. By not accepting certain inputs, sociable interfaces express the assumption that the environment never generates these inputs: hence, sociable interfaces (like other interface models) model both the output behavior, and the input assumptions, of a component. This approach implies a notion of composition (based on synthesizing the weakest environment assumptions that guarantee compatibility) which is not present in the I/O Automata Model.

Interface models are the subject of many recent works. Previous interface models, such as interface automata [6, 8] and interface modules [7, 3] were based on either actions, or variables, but not both. Sociable interfaces do not break new ground in the conceptual theory of interface models. However, by allowing both actions and variables, they take advantage of the existing models and try to avoid their deficiencies. The rest of this section is devoted to a quick presentation of existing interface models.

Variable-based interface formalisms. In variable-based interface formalisms, such as the formalisms of [7, 3], communication is mediated by input and output variables, and the system evolves in synchronous steps. It is well known that synchronous, variable-based models can also encode communication via actions [1]: the generation of an output $a!$ is translated into the toggling of the value of an (output) boolean variable x_a, and the reception of an input $a?$ is encoded by forcing a transition to occur whenever the (input) variable x_a is toggled. This encoding is made more attractive by syntactic sugar [1]. However, this encoding prevents the modeling of many-to-one and many-to-many communication.

In fact, due to the synchronous nature of the formalism, a variable can be modified at most by one module: if two modules modified it, there would be no simple way to determine its updated value.[6] Since the generation of an output $a!$ is modeled by toggling the value of a boolean variable x_a, this limitation indicates that an output action can be emitted at most by one module. As a consequence, we cannot write modules that can accept inputs from multiple sources: every module must know precisely which other modules can provide inputs to it, so that distinct communication actions can be used. The advance knowledge of the modules involved in communication hampers module re-use.

Action-based interface formalisms. Action-based interfaces, such as the models of [6, 5, 8], enable a natural encoding of asynchronous communication. In previous proposal, however, two interfaces could be composed only if they did not share output actions — again ruling out many-to-one communication.

[6] A possible way out would be to define that, in case of simultaneous updates, only one of the updates occurs nondeterministically. This choice, however, would lead to a complex semantics, and to complex analysis algorithms.

Furthermore, previous action-based formalisms lacked a notion of global variables which are visible to all the modules of a system. Such global variables are a very powerful and versatile modeling paradigm, providing a notion of global, shared state. Mimicking global variables in purely action-based models is rather inconvenient: it requires encapsulating every global variable by a module, whose state corresponds to the value of the variable. Read and write accesses to the variable must then be translated to appropriate sequences of input and output actions, leading to cumbersome models.

References

1. R. Alur and T.A. Henzinger. Reactive modules. *Formal Methods in System Design*, 15:7–48, 1999.
2. R. Alur, T.A. Henzinger, O. Kupferman, and M.Y. Vardi. Alternating refinement relations. In *CONCUR 98: Concurrency Theory. 9th Int. Conf.*, volume 1466 of *Lect. Notes in Comp. Sci.*, pages 163–178. Springer-Verlag, 1998.
3. A. Chakrabarti, L. de Alfaro, T.A. Henzinger, and F.Y.C. Mang. Synchronous and bidirectional component interfaces. In *CAV 02: Proc. of 14th Conf. on Computer Aided Verifi cation*, volume 2404 of *Lect. Notes in Comp. Sci.*, pages 414–427. Springer-Verlag, 2002.
4. K.M. Chandy and J. Misra. *Parallel Program Design: A Foundation*. Addison-Wesley Publishing Company, 1988.
5. L. de Alfaro. Game models for open systems. In *Proceedings of the International Symposium on Verifi cation (Theory in Practice)*, volume 2772 of *Lect. Notes in Comp. Sci.* Springer-Verlag, 2003.
6. L. de Alfaro and T.A. Henzinger. Interface automata. In *Proceedings of the 8th European Software Engineering Conference and the 9th ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE)*, pages 109–120. ACM Press, 2001.
7. L. de Alfaro and T.A. Henzinger. Interface theories for component-based design. In *EMSOFT 01: 1st Intl. Workshop on Embedded Software*, volume 2211 of *Lect. Notes in Comp. Sci.*, pages 148–165. Springer-Verlag, 2001.
8. L. de Alfaro and T.A. Henzinger. Interface-based design. In *Engineering Theories of Software Intensive Systems, proceedings of the Marktoberdorf Summer School*. Kluwer, 2004.
9. L. de Alfaro and M. Stoelinga. Interfaces: A game-theoretic framework to reason about open systems. In *FOCLASA 03: Proceedings of the 2nd International Workshop on Foundations of Coordination Languages and Software Architectures*, 2003.
10. Xavier Leroy. Objective caml. http://caml.inria.fr/ocaml/index.en.html.
11. N.A. Lynch. *Distributed Algorithms*. Morgan-Kaufmann, 1996.
12. R.I. Bahar, E.A. Frohm, C.M. Gaona, G.D. Hachtel, E. Macii, A. Pardo, and F. Somenzi. Algebraic Decision Diagrams and Their Applications. In *IEEE /ACM International Conference on CAD*, pages 188–191, Santa Clara, California, 1993. IEEE Computer Society Press.
13. Fabio Somenzi. Cudd: Cu decision diagram package. http://vlsi.colorado.edu/~fabio/CUDD/cuddIntro.html.
14. A. Srinivasan, T. Kam, S. Malik, and R. Brayton. Algorithms for discrete function manipulation. In *Proceedings International Conference CAD (ICCAD-91)*, 1990.