

# On the Design and the Implementation of a Game-based Model for Open Systems: Current Status and Perspectives.\*

Marco Faella<sup>1</sup> and Axel Legay<sup>2</sup>

<sup>1</sup> Dipartimento di Scienze Fisiche, Università di Napoli “Federico II”, Italy

<sup>2</sup> Department of Computer Science, University of Liège, Belgium

**Abstract.** Sociable interfaces are a game-based model with rich communication primitives that facilitate the modeling of software and distributed systems. The first model of sociable interfaces has been introduced in [9], and then implemented in a tool called TICC [2]. While the primary goal of TICC was to perform the composition of two or more sociable interfaces, it has now evolved into a complete specification and verification tool. This paper presents the current status of the sociable interfaces model and of TICC. It discusses the improvements that have been brought to previous versions of the model and the tool, and investigates future directions of research.

## 1 Introduction

The trend in software and system engineering is towards component-based design: systems are designed by combining small components into bigger ones. Components offer thus the unit in which complex design problems can be decomposed, allowing the reduction of a single complex design problem into smaller design problems, more manageable in complexity, that can be solved in parallel. Components also provide a unit of reuse, defining the boundaries in which functionality can be packaged, documented and reused.

Components are designed to work as parts of larger systems: they make assumptions on their environment, and they expect that these assumptions will be met in the actual environment. In other words, a component is typically an open system which has some free inputs provided by other components and which in turn provides inputs to other components. It is thus obvious that the effective reuse of software requires adequate documentation of the component’s behavior and the conditions under which it can be used, along with methods for checking that components are assembled in an appropriate way. Such documentation is commonly referred to as the *interface* of the component.

There have been many works on the design and implementation of good interfaces for components. Most of those works focus on capturing the *data dimension* of interfaces (“What are the value constraints on data communicated between components?”) [22].

In a series of recent works [5, 14, 12], de Alfaro and Henzinger introduced *interface theories* that is as a formal notion of component interfaces that uses games to represent the interaction between the behavior originating within a component and the behavior originating from the component’s environment.

This game-based model is able to capture the *protocol dimension* of interfaces (“What are the temporal ordering constraints on communication events between components?”) which makes it similar to a type system: indeed, it could be termed a “behavioral” type system for component interaction (see [17]).

In recent work [9], we introduced sociable interfaces — a concrete model of interface theories with rich communication primitives and design facilities. The theory of sociable interfaces was first presented in [9], and then implemented in a tool called TICC. The preliminary version of TICC was described in [2], but the tool (and so the model) has now been substantially improved<sup>3</sup>.

---

\* Axel Legay is supported by a F.R.I.A grant

<sup>3</sup> The first version of the tool was limited to the composition of interfaces specified with a restrictive input language.

The first goal of this paper is to present the current version of the sociable interface model and of TICC, as well as to discuss the perspectives for future work. The second goal is to make the link between the theory behind the tool and what the user really needs to know to use it.

Due to space limitations, we complete the practical part of the paper with a technical report [11].

## 2 Open Systems and Games, Where is the Link?

In this section we sketch the main features of the interface theories approach and we describe the link with game theory. The reader is referred to [8, 15] for more details.

In interface theories, an interface support the component-based design in the following ways.

**Interface specification.** An interface specifies how a component interacts with its environment. It describes the input assumptions that the component makes on the environment (methods that can be called) and the output guarantees it provides (methods calls, ...). Interfaces capture the I/O behavior of a component by an automaton whose syntax is similar to the I/O automata of [22]. Unlike traditional models, including I/O automata, that at every state must be receptive to every possible input event, in interface theories it is possible that inputs are illegal (cannot be accepted) at some states. One of the main advantages of making explicit assumptions about the environment is that it gives rise to an optimistic compatibility test when interfaces are composed, as explained below. Moreover, from a practical point of view, the ability to forbid inputs removes the need to specify “what happens” when taking an undesirable input. On the other hand, it is important to guarantee that the interface works in at least one environment, i.e. that it is well-formed. An interface is naturally modeled as a game between the players Input and Output. Input represents the environment: the moves of Input represent the inputs accepted from the environment. Output represents the component: the moves of Output represent the possible outputs generated by the component. Then, an interface is well-formed (i.e. can work in at least one environment) if the Input player has a winning strategy in the game, which means that the environment can meet all input assumptions.

**Interface Composition.** Like most existing models, interfaces interact through the synchronization of common input and output events. The interpretation of inputs and outputs as assumptions and guarantees, respectively, implies that, when composing two interfaces  $P$  and  $Q$ , we have to ensure that  $P$ 's output guarantees satisfy  $Q$ 's input assumptions and vice versa. Concretely, consider the two interfaces  $P$  and  $Q$ , in one state of the composition. If  $P$  wants to emit an output that cannot be accepted by  $Q$  in that state (i.e. an output guarantee that violates an input assumption), then a *local incompatibility* occurs. While many approaches would be pessimistic and consider the two interfaces to be incompatible, the interface approach is optimistic, by assuming that the environment will steer away from locally incompatible states. Thus, two interfaces are *compatible* if there exists an environment to use the components together, and ensure that the assumptions of both are met. Interface composition thus consists in synthesizing the most liberal input strategy in the composite system that avoids all locally incompatible states. This can be done by classical game-theoretic algorithms [13].

**Interface Refinement and Model Checking.** Composition is certainly the most important operation that differentiates the theory of interfaces from other approaches. However it is also worth mentioning that, since the model distinguishes between Inputs and Outputs, the notion of refinement [21] naturally reduces to the one of alternating simulation [4] between the players of the components that have to be compared. Moreover, while this game-based model do not forbid model checking open systems with classical closed system logics such as branching time temporal logic CTL [7], it also offers the possibility of using logics that have been designed for open systems, such as alternating temporal logic [3].

As pointed out in [8], most of the ideas on which the approach of interface theories is based have

been introduced earlier in the literature. The idea of specifying independent constraints for input and output behavior is present in trace theories [16]. All the algorithms used to solve the game are standard [4]. Refinement is based on alternating simulation [4]. Checking compatibility of composition can in certain cases be solved by controller synthesis [27]. Thus, the novelty in the game-based approach for open systems is not in any isolated algorithmic aspect, but rather in the recognition that refinement, composition, and synthesis can be homogeneously cast as game relations and problems. Note that the work done on game semantics for process algebra [25] and programming languages [1] also suggests that games provide an unifying framework for interaction between components. A comparison between those models and ours has been given in [17].

### 3 Before Sociable Interfaces

There have been many recent works on the design of concrete models of interface theories. They can be divided into two categories: (1) interface automata [12] that are asynchronous models that communicate via disjoint<sup>4</sup> input and output actions, and (2) interface modules [5] that are synchronous models that communicate via disjoint input and output variables.

While those models clearly show the applicability of the approach, they impose drastic restrictions on the way that components can communicate together. As an example, due to the distinction between the names of input and output actions (resp. variables), none of those models allow several components to communicate with a third one using the same action (resp. variable) name. Moreover, the fact that actions and variables cannot be combined introduces difficulties to the design of many interesting systems, such as those that manipulate global resources.

### 4 The Sociable Interface Model: Theory

This section gives a theoretical description of the sociable interface model. We assume a fixed set  $\mathcal{V}$  of variables that are interpreted over a given domain  $\mathcal{D}$ . Given  $V \subseteq \mathcal{V}$ , a *state* over  $V$  is a mapping  $s : V \rightarrow \mathcal{D}$  and  $\llbracket V \rrbracket$  denotes the set of all states over  $V$ . For a set of variables  $U \subseteq V$ , and a state  $s \in \llbracket V \rrbracket$ , the projection of  $s$  on  $U$  is a state  $t \in \llbracket U \rrbracket$  denoted as  $s[U]$ . For two disjoint sets of variables  $V_1$  and  $V_2$ , and two states  $s_1 \in \llbracket V_1 \rrbracket$  and  $s_2 \in \llbracket V_2 \rrbracket$ , the operation  $(s_1 \circ s_2)$  composes the two states resulting in a new state  $s = s_1 \circ s_2 \in \llbracket V_1 \cup V_2 \rrbracket$ , such that  $s(x) = s_1(x)$  for all  $x \in V_1$  and  $s(x) = s_2(x)$  for all  $x \in V_2$ .

Given a set  $V$  of variables, we denote by  $Preds(V)$  the set of first-order predicate formulas with free variables in  $V$ ; for the moment, we assume that these predicates are written in some specified first-order language. We let  $V' = \{x' \mid x \in V\}$  be the set of primed versions of variables in  $V$ . Intuitively, a variable  $x' \in V'$  represents the *next value* of  $x \in V$ . Given a formula  $\psi \in Preds(V)$  and a state  $s \in \llbracket V \rrbracket$ , we write  $s \models \psi$  if the predicate formula  $\psi$  is true when its free variables are interpreted as specified by  $s$ . Given a formula  $\rho \in Preds(V \cup V')$  and two states  $s, s' \in \llbracket V \rrbracket$ , we write  $\langle s, s' \rangle \models \rho$  if the formula  $\rho$  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  $\psi'$  the predicate obtained by substituting  $x$  by  $x'$  in  $\psi$ , for all  $x \in V$ .

With those definitions, we can define a sociable interface as follows:

**Definition 1.** A *sociable interface* is a tuple  $M = (Act^G, Act^L, \mathcal{D}, V^G, V^L, V^H, W, \rho^{IL}, \rho^{IG}, \rho^O, \rho^L, \psi^I, \psi^O, I)$ .  $Act^G$  is a set of set of *global actions*, and  $Act^L$  is a set of *local actions*. We set  $Act = Act^G \cup Act^L$ .  $V^G$  is a set of *global variables*,  $V^L$  is a set of *local variables*, and  $V^H \subseteq V^G$  is a set of

<sup>4</sup> This separation was mainly motivated to keep the distinction between the Input and the Output players.

*history variables*. We require  $V^L \cap V^G = \emptyset$ . We set  $V^{\text{all}} = V^L \cup V^G$  and  $V = V^L \cup V^H$ . All variables are interpreted over the domain  $\mathcal{D}$ . For each action  $a \in \text{Act}$  we denote by  $W(a)$  the set of variables that can be modified by  $a$ . For each  $a \in \text{Act}^G$ , the predicates  $\rho^{LL}(a) \in \text{Preds}(V^{\text{all}} \cup (V^{\text{all}})')$ ,  $\rho^{IG}(a) \in \text{Preds}(V^{\text{all}} \cup (V^G)')$ , and  $\rho^O(a) \in \text{Preds}(V^{\text{all}} \cup W(a)')$  are respectively the input local, the input global, and the output transition predicates for  $a$ . We require  $\rho^{LL}(a)$  to be *deterministic* w.r.t. variables in  $V^L$ , that is, for all  $a \in \text{Act}^G$ , all  $s \in \llbracket V^{\text{all}} \rrbracket$ , and all  $t \in \llbracket (V^G)' \rrbracket$ , there is a unique  $u \in \llbracket (V^L)' \rrbracket$  such that  $s \circ t \circ u \models \rho^{LL}(a)$ . We let  $\rho^I(a) = \rho^{LL}(a) \wedge \rho^{IG}(a)$ . For each  $a \in \text{Act}^L$ , the predicate  $\rho^L(a) \in \text{Preds}(V^{\text{all}} \cup (V^L)')$  is the *local transition predicate* for  $a$ . Local transitions represent internal choices made by the component that cannot be viewed from the outside world. We define  $\psi^O \in \text{Preds}(V^{\text{all}})$  and  $\psi^I \in \text{Preds}(V^{\text{all}})$  to be respectively the output and input *invariant predicates* of  $M$ . Finally, we define  $I \in \text{Preds}(V^{\text{all}})$  to be the predicate that characterizes the set of *initial states* of  $M$ .

This model differs from the one given in [9] by the addition of local actions and of an initial condition. The initial condition is the prelude announcement to a new feature of the tool: the manipulation of sets of states.

Sociable interfaces do not distinguish between input and output actions/variables. Rather, they 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. In the model, one assumes that output and local transitions are the only responsible for the updates of the value of the variables. The global part of an input transition — which represents an input assumption — only makes assumptions on the value of the global variables. The local part of an input transition, instead, allows the interface to immediately react to the reception of an action from the environment, subject to two restrictions: First, the interface can only modify the value of its local variables, and second, it must do so in a *deterministic* fashion. These restrictions are necessary to ensure that each state change is driven by the component issuing the output action. The component which is receiving the action can express assumptions on how the action will update the global variables, but it cannot participate in the choice of the new values.

We denote  $S = \llbracket V^{\text{all}} \rrbracket$  to be the set of states of  $M$ . As a shorthand, we let  $\varphi^I = \{s \in S \mid s \models \psi^I\}$ ,  $\varphi^O = \{s \in S \mid s \models \psi^O\}$ . We define the restriction of transitions w.r.t. invariants, i.e.  $\widehat{\rho}^I(a) = \rho^I(a) \wedge (\psi^I)'$ ,  $\widehat{\rho}^O(a) = \rho^O(a) \wedge (\psi^O)'$  and  $\widehat{\rho}^L(a) = \rho^L(a) \wedge (\psi^O)' \wedge \text{Unchgd}(V^{\text{all}} \setminus V^L)$ .

Note that variables whose next value is not specified in  $\rho^O$  (resp.  $\rho^L$ ) are assumed to keep their value in  $\widehat{\rho}^O$  (resp.  $\widehat{\rho}^L$ ). The latter is reasonable since output (resp. local) transitions reflect the behaviors of the component. However, no assumption is made on  $\rho^I$  and  $\widehat{\rho}^I$ . The reason is that  $\rho^I$  represents the behaviors of the environment.

#### 4.1 The Game under the Model

In this section, we assume a sociable interface  $M = (\text{Act}^G, \text{Act}^L, \mathcal{D}, V^G, V^L, V^H, W, \rho^{LL}, \rho^{IG}, \rho^O, \rho^L, \psi^I, \psi^O, I)$ , and we describe its semantics in term of a turn-based game between the two players Input and Output.

The game is played in an arena that is  $S = \llbracket V^{\text{all}} \rrbracket$ . At each round, from the current state, both players simultaneously choose a move that defines the next state of the game.

**Definition 2 (Moves).** The sets  $\Gamma^I(M, s)$  and  $\Gamma^O(M, s)$  of Input and Output moves at  $s \in S$  are defined as follows:

$$\begin{aligned} \Gamma^I(M, s) &= \{\Delta_0\} \times \{s' \in \llbracket V^{\text{all}} \rrbracket \mid s'[V] = s[V]\} \cup \\ &\quad \{\langle a, s' \rangle \in \text{Act}^G \times \llbracket V^{\text{all}} \rrbracket \mid \langle s, s' \rangle \models \widehat{\rho}^I(a)\} \\ \Gamma^O(M, s) &= \{\Delta_0\} \cup \{\langle a, s' \rangle \in \text{Act}^G \times \llbracket V^{\text{all}} \rrbracket \mid \langle s, s' \rangle \models \widehat{\rho}^O(a)\} \cup \\ &\quad \{\langle a, s' \rangle \in \text{Act}^L \times \llbracket V^{\text{all}} \rrbracket \mid \langle s, s' \rangle \models \widehat{\rho}^L(a)\}. \end{aligned}$$

The  $\{\Delta_0\}$  is an extra move to ensure that both players have a move to propose in each state. Note that, when Input plays the move  $\Delta_0$ , it 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 interface.

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  $s \in S$  and moves  $m^I \in \Gamma^I(M, s)$  and  $m^O \in \Gamma^O(M, s)$ , the outcome  $\delta(M, s, m^I, m^O) \subseteq S$  of playing  $m^I$  and  $m^O$  at  $s$  can be defined as follows.

$$\begin{aligned} \delta(M, s, \langle \Delta_0, s' \rangle, \Delta_0) &= \{s'\}, & \delta(M, s, \langle \Delta_0, s' \rangle, \langle a, t' \rangle) &= \{s', t'\}, \\ \delta(M, s, \langle a, s' \rangle, \Delta_0) &= \{s'\}, & \delta(M, s, \langle a, s' \rangle, \langle b, t' \rangle) &= \{s', t'\}. \end{aligned}$$

The move outcomes show the priority that a “real” move has on the output  $\Delta_0$  move, but not on the input  $\Delta_0$  move. For  $s \in S$ , we define the set of runs starting from  $s$  as the set  $Runs(M, s) \subseteq S^\omega$  of all infinite sequences  $s_0 s_1 s_2 \dots$ , such that  $s_0 = s$ , and for all  $i \geq 0$ ,  $s_{i+1} \in \delta(M, s_i, m^I, m^O)$ , for some  $m^I \in \Gamma^I(M, s_i)$ ,  $m^O \in \Gamma^O(M, s_i)$ . We also set  $Runs(M) = \bigcup_{s \in S} Runs(M, s)$ . Given  $i \geq 0$ , The finite prefix  $\sigma_{0:i}$  of a run  $\sigma = s_0 s_1 s_2 \dots s_i \dots$  is the finite sequence  $s_0 s_1 s_2 \dots s_i$  that is constituted of the  $i + 1$  first states of  $\sigma$ . A strategy for player  $p \in \{I, O\}$  is a function  $\pi^p$  that for each run  $\sigma \in Runs(M)$  associates a move  $\pi^p(\sigma_{0:i}) \in \Gamma^p(M, s_i)$  to each finite prefix  $\sigma_{0:i}$  of  $\sigma$ . We denote by  $\Pi_M^I$  and  $\Pi_M^O$  the set of input and output strategies for  $M$ , respectively. Let  $\pi^I \in \Pi_M^I$  and  $\pi^O \in \Pi_M^O$ , the set  $\hat{\delta}(M, s, \pi^I, \pi^O)$  of  $\pi^I$  and  $\pi^O$  from  $s$  consists of all runs  $\sigma = s_0 s_1 s_2 \dots$  such that  $s = s_0$ , and for all  $i \geq 0$ ,  $s_{i+1} \in \delta(M, s_i, \pi^I(\sigma_{0:i}), \pi^O(\sigma_{0:i}))$ . Given a state  $s \in S$  and a goal  $\gamma \subseteq Runs(M, s)$ , we say that  $s$  is winning for input (resp. output) with respect to  $\gamma$ , and we write  $s \in Win^I(M, \gamma)$  (resp.  $s \in Win^O(M, \gamma)$ ), iff there is  $\pi^I \in \Pi_M^I$  (resp.  $\pi^O \in \Pi_M^O$ ) such that for all  $\pi^O \in \Pi_M^O$ ,  $\hat{\delta}(M, s, \pi^I, \pi^O) \subseteq \gamma$  (resp.  $\pi^I \in \Pi_M^I$ ,  $\hat{\delta}(M, s, \pi^I, \pi^O) \subseteq \gamma$ ).

**Definition 4 (Normal Form).** [9] We say that  $M$  is in normal form iff  $\varphi^I = Win^I(M, \square\varphi^I)$ , and  $\varphi^O = Win^O(M, \square\varphi^O)$ , where  $\square X = \{s_0 s_1 s_2 \dots \in Runs(M) \mid \forall i \geq 0. s_i \in X \subseteq S\}$ .

The computation of  $Win^I(M, \square X)$  is referred to a safety game whose objective is  $\square X$ .

Definition 4 induces the trivial, but important lemma.

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

$$\begin{aligned} \forall s \in \varphi^I. \forall (a, s') \in \Gamma^O(M, s). s' \in \varphi^I, \\ \forall s \in \varphi^O. \forall (a, s') \in \Gamma^I(M, s). s' \in \varphi^O. \end{aligned}$$

**Definition 5 (Well-formed Sociable Interface).** We say that  $M$  is well-formed iff (1) it is in normal form, and (2)  $\varphi^I \cap \varphi^O \cap \{s \in S \mid s \models I\} \neq \emptyset$ .

Note that point (2) of the definition ensures that  $M$  is well-formed only if the states that are well-formed can be reached from its initial states.

## 5 The Sociable Interface Model: Practice

The sociable interface model of Section 4 has been implemented in a tool called TICC (Tool for Interface Compatibility and Composition) The documented code of TICC is freely available and can be downloaded from <http://dvlab.cse.ucsc.edu/dvlab/Ticc>. This website is a Wiki that also contains the documentation for the tool, and several additional examples. TICC allows users to specify sociable interfaces, called “modules”, using a textual language based on guarded commands, perform operations on the modules, and verify

properties of modules using functions that extend the capabilities of the OCaml [20] command-line. In the rest of the section, we describe the internal representation used by the tool. The next section outlines how the tool is used.

In the rest of this Section we consider a sociable interface  $M = (Act^G, Act^L, \mathcal{D}, V^G, V^L, V^H, W, \rho^{IL}, \rho^{IG}, \rho^O, \rho^L, \psi^I, \psi^O, I)$ . Internally, TICC relies on a symbolic representation of sociable interfaces based on MDDs [24, 26] that are used to represent the predicates  $\rho^{IL}, \rho^{IG}, \rho^O, \rho^L, \psi^I, \psi^O$  and  $I$ . MDDs are graph-like data structures that allow to represent and manipulate functions of the type  $A \rightarrow \{\text{T}, \text{F}\}$  (i.e. predicates over  $A$ ), for a finite set  $A$  of variables whose domain range over the Boolean and the bounded integers. It is well known that MDDs are a very compact representation on which Boolean operations and quantifier elimination can be performed efficiently. By abuse of notation, given a predicate  $X \in \text{Preds}(V^{\text{all}})$ , we denote by  $X$  also the set of states  $\{s \in S \mid s \models X\}$ . However, in TICC, all the algorithms are implemented with MDDs that represent set of states.

We now consider the implementation of the first operation we need to perform on an interface, i.e. checking well-formedness. As mentioned in Section 4, this operation reduces to the computation of  $\text{Win}^p(M, \square\varphi)$  for  $p \in \{I, O\}$ . It is well known (see [4]) that the set of winning states can be characterized as a fixpoint of an operator involving the so-called *controllable predecessors operators*  $Cpre^I(\cdot)$  and  $Cpre^O(\cdot)$ .

**Definition 6 (Controllable Predecessor Operator).** For all predicates  $X \in \text{Preds}(V^{\text{all}})$ , we have:

$$\begin{aligned} Cpre^I(X) &= \exists m^I \in \Gamma^I(M, s) . \forall m^O \in \Gamma^O(M, s) . \forall t \in \delta(M, s, m^I, m^O) . t \models X \\ Cpre^O(X) &= \exists m^O \in \Gamma^O(M, s) . \forall m^I \in \Gamma^I(M, s) . \forall t \in \delta(M, s, m^I, m^O) . t \models X \end{aligned}$$

Intuitively,  $Cpre^I(X) \in \text{Preds}(V^{\text{all}})$  (resp.  $Cpre^O(X) \in \text{Preds}(V^{\text{all}})$ ) is a predicate that holds true for each state  $s \in S$  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. We have the following definition.

For all  $\varphi \in \text{Preds}(V^{\text{all}})$ , we have  $\text{Win}^I(M, \square\varphi) = \nu X . [\varphi \wedge Cpre^I(X)]$ , and  $\text{Win}^O(M, \square\varphi) = \nu X . [\varphi \wedge Cpre^O(X)]$ , where  $\nu X . f(X)$  denotes the greatest fixpoint of the operator  $f$ . Since  $Cpre^I(\cdot)$  (resp.  $Cpre^O(\cdot)$ ) is monotonic, the fixpoints exist and can be computed by Picard iteration, i.e.  $X_0 = \varphi, X_{i+1} = \varphi \wedge Cpre^I(X_i), \dots$ , and  $X_n = X_{n+1} = \text{Win}^I(M, \square\varphi)$ . Due to the definition of the moves and move outcomes of our game, it is possible to apply reasoning that is identical to the one given in [9], and obtain the following result.

**Lemma 2.** For all predicates  $\varphi \in \text{Preds}(V^{\text{all}})$ , we have

$$\text{Win}^I(M, \square\varphi) = \nu X . [\varphi \wedge \forall Pre^O(X)], \text{ and } \text{Win}^O(M, \square\varphi) = \nu X . [\varphi \wedge \forall Pre^I(X)],$$

where

$$\begin{aligned} \forall Pre^O(X) &= \bigwedge_{a \in Act^G} \forall (V^{\text{all}})' . (\widehat{\rho}^O(a) \Rightarrow X') \wedge \bigwedge_{a \in Act^L} \forall (V^{\text{all}})' . (\widehat{\rho}^L(a) \Rightarrow X') \\ \forall Pre^I(X) &= \bigwedge (\exists (V^{\text{all}})' . X' \wedge \text{Unchgd}(V)) \wedge \bigwedge_{a \in Act^G} \exists (V^{\text{all}})' . (\widehat{\rho}^I(a) \wedge X'). \end{aligned}$$

Note that  $\forall Pre^O(X) \in \text{Preds}(V^{\text{all}})$  and  $\forall Pre^I(X) \in \text{Preds}(V^{\text{all}})$ .

It is possible to obtain several algorithms to compute the fixpoints of  $\forall Pre^O(X)$  and of  $\forall Pre^I(X)$  simply by reorganizing the Picard iteration. Some of those algorithms have been implemented and documented in the tool.

## 5.1 An Introduction to the Use of TICC

This section is a summarized introduction to the use of TICC. More detailed examples that illustrate the full input language of the tool are given in [11].

We illustrate the modeling language of TICC by means of a very simple example: a fire detection system that is composed of a control unit and several smoke detectors. When a detector senses smoke (input *smoke*), it reports it by emitting an output *fire*. When the control unit receives the input *fire* from any of the detectors, it emits the output *call\_fd*, corresponding to a call to the fire department. Additionally, an input *disable* disables both the control unit and the detectors, so that the smoke sensors can be tested without triggering an alarm.

Below, we provide the code for the control unit module (`ControlUnit`) and for one of the (several) fire detectors (`FireDetector1`):

```
module ControlUnit:
  var s: [0..3] // 0=waiting, 1=alarm raised, 2=fd called, 3=disabled
  initial : s = 0
  iinv : true
  ooinv : true
  input fire:      { local: s = 0 | s = 1 ==> s' := 1
                    else s = 2 ==>           }
  input disable: { local: true ==> s' := 3 }
  output call_fd: { s = 1 ==> s' = 2 }
endmodule

module FireDetector1:
  var s: [0..2] // 0=idle, 1=smoke detected, 2=inactive
  initial : s = 0
  iinv : true
  ooinv : true
  input smoke1: { local: s = 0 | s = 1 ==> s' := 1
                 else s = 2 ==>           } // do nothing if inactive
  output fire:  { s = 1 ==> s' = 2 }
  input fire:   { } // other modules can detect fire too
  input disable: { local: true ==> s' := 2 }
endmodule
```

The body of each module starts with the (possibly empty) list of its local variables (in the example, those variables are used to encode the locations of the modules). This list is followed by the declarations of the initial condition of the module and by its input and output invariants. Note that both the declaration of the initial condition and of the invariants can be omitted, in which case they are automatically set to true. The transitions are specified using guarded commands *guard*  $\Rightarrow$  *command*, where *guard* and *command* are Boolean expressions over the local and global variables; as usual, primed variables refer to the values after a transition is taken. For instance, the output transition *fire* in module `FireDetector1` can be taken only when the local variable *s* has value 1, and it leads to a state where *s* = 2. When a transition starts with the keywords *input* or *output*, then the associated action is automatically considered to be global. The variables whose next value is not precised by the user are updated following the theory given in Section 4. We now go through the details of parsing a TICC program. We consider the fire detection systems and we suppose that it is given in a file whose name is `fire.si`. First the user has to invoke TICC using the command `ticc` from the shell. The result of this operation is an OCaml prompt from where one must type: “open Ticc;;”. At this point the functions in the module of TICC become available at the top level (these functions are documented in the file `ticc/doc/api/Ticc.html` and in [11]). Next, one has to read a TICC program from a file, here `fire.si`. This is achieved by typing the command “parse “fire.si”;;”. The `parse` function reads in a `.si` file describing modules and possibly global variables, and places these

definition into a global namespace. If the .si file does not follow the syntax of the input language, the function reports an appropriate error message. After parsing a TICC program, it is possible to construct the symbolic internal representation of each module. As an example, here is the command to construct the representation of ControlUnit: “let controlunit = mk\_sym "ControlUnit";”. The command mk\_sym constructs MDDs representations for all the predicates of the module. Moreover, it plays a safety game and restricts the invariants to ensure that the module is well-formed. Thus, there can be a difference between the module specified by the user and the one that will be used by the tool. Such differences can be detected with printout functions of TICC. As an example, the following command can be used to print out the content of module ControlUnit: “print\_symmod controlunit;”. The printout functions are particularly useful for debugging in TICC. In the new release of the tool, we also added a random simulation function on symbolic modules. This function generates an HTML file with the result of the simulation (see Example 2). This is particularly useful in the early stages of model construction, to confirm that the model behaves as intended. Finally, let us note that one can also write *script files* for TICC. A script file is a file that groups a set of commands that can be executed in one step. One can invoke TICC to execute the script file with the following command from the shell prompt: “ticc scriptfile”.

## 6 Composing Sociable Interfaces

In this section, we consider the composition of two sociable interfaces  $M_1$  and  $M_2$ , where  $M_i = (Act_i^G, Act_i^L, \mathcal{D}, V_i^G, V_i^L, V_i^H, W_i, \rho_i^{LL}, \rho_i^{IG}, \rho_i^O, \rho_i^L, \psi_i^I, \psi_i^O, I_i)$ . In sociable interfaces, composition is done in four steps. The theory behind those steps has already been given in [9] and the extension to our model is immediate. Here, we mainly focus on what the user really needs to know when performing the operation in TICC.

### 6.1 The Composability Condition

First, we need to check whether  $M_1$  and  $M_2$  are composable. This requires checking that if an action  $a \in Act_1^G$  (respectively  $Act_2^G$ ) of  $M_1$  (resp.  $M_2$ ) has an output transition that can modify a history variable of  $M_2$  (resp.  $M_1$ ), then  $M_2$  (resp.  $M_1$ ) has an input transition for action  $a \in Act_2^G$  (resp.  $a \in Act_1^G$ ). This check is the main motivation for distinguishing between history and history-free variables: an interface should only know all actions of other interfaces that modify its history variables. If we dropped the distinction, then an interface would have to know all actions of other interfaces that can change any of its variables, and this could greatly increase its number of transitions. Note that, in the new version of the tool, we implemented a pattern matching mechanism that can substantially reduce the number of transitions. As an example:

```
input a* : {implementation}
```

is equivalent to say that a module accepts all the input transitions whose first letter is “a”. The utility of this mechanism is illustrated in the “house example” of [11].

### 6.2 The product

If the two interfaces are compatible, then we can define their product. The product of  $M_1$  and  $M_2$  is a sociable interface  $M_{12} = (Act_{12}^G, Act_{12}^L, \mathcal{D}, V_{12}^G, V_{12}^L, V_{12}^H, W_{12}, \rho_{12}^{LL}, \rho_{12}^{IG}, \rho_{12}^O, \rho_{12}^L, \psi_{12}^I, \psi_{12}^O, I_{12})$ . First, we have  $Act_{12}^G = Act_1^G \cup Act_2^G$ ,  $Act_{12}^L = Act_1^L \cup Act_2^L$ ,  $V_{12}^G = V_1^G \cup V_2^G$ ,  $V_{12}^L = V_1^L \cup V_2^L$ ,  $V_{12}^H = V_1^H \cup V_2^H$ ,  $\psi_{12}^I = \psi_1^I \wedge \psi_2^I$ ,  $\psi_{12}^O = \psi_1^O \wedge \psi_2^O$ , and  $I_{12} = I_1 \wedge I_2$ . The transitions of  $M_{12}$  are combinations of the transitions of  $M_1$  and  $M_2$ . For each shared global action, the output transition of  $M_1$  (resp.  $M_2$ ) synchronizes (in our model, synchronization boils down to conjoining the corresponding predicates) with the *local part* of the input transition of  $M_2$  (resp.  $M_1$ ), and gives rise to output transitions in the product. The reason not to synchronize with the global part of the input is to ensure that only output transitions can modify the values of the global

variables (inputs are only supposed to make assumptions on them). The input transitions of  $M_1$  and  $M_2$  corresponding to the same shared global actions are also synchronized, and lead to an input transition in the product. Finally, the interfaces interleave asynchronously on transitions labeled by non-shared global actions, and on local actions.

### 6.3 Locally Incompatible States

The product  $M_{12}$  can contain locally incompatible states in which one of the interfaces being composed wants to issue an output transition labeled by a shared global action, while the other interface does not have a corresponding (same action name) global input transition from that state which agrees with the output transition on the updates of global variables. We denote by *Good* the set of locally compatible states.

### 6.4 Synthesizing a Strategy

After computing  $M_{12}$  and *Good*, the next operation is to compute the set of states *Win* from which the Input player of  $M_{12}$  has a strategy to always stay in *Good*. In other words, we play a safety game whose arena is the set of states of the product, and whose objective is the set *Good*. The set *Win* is used to restrict the input invariant and the initial condition of  $M_{12}$  which is equivalent to restrict the environments in where it can be used. This is an optimistic approach, since two interfaces are considered to be compatible if they can work in at least one environment.

### 6.5 Implementation

The implementation of the composition operation is direct since it only requires Boolean combinations of predicates (for checking the compatibility condition, for computing  $M_{12}$ , and for computing the predicate representing the set *Good*) and the solution of a safety game (for computing *Win*) (see Section 5).

The composition of two compatible interfaces  $M_1$  and  $M_2$  is denoted by  $M_1 \parallel M_2$ . We have the following theorem.

**Theorem 1.** *For all sociable interfaces  $M_1$ ,  $M_2$ , and  $M_3$ , either both  $(M_1 \parallel M_2) \parallel M_3$  and  $M_1 \parallel (M_2 \parallel M_3)$  are undefined, because some of the interfaces are not compatible, or  $(M_1 \parallel M_2) \parallel M_3 = M_1 \parallel (M_2 \parallel M_3)$ .*

Theorem 1 shows that sociable interfaces support incremental design, i.e. the components can be composed and checked in any order.

### 6.6 The Composition Operation in TICC

In TICC, the user performs composition with the function “compose” followed by the name of two symbolic modules. The result of this call is either a new symbolic module, or an error message if the two modules are not compatible.

*Example 1.* Consider the fire detection system of Section 5.1, and suppose also the existence of a faulty detector `Wrong_FireDetector2` which does not react on input `disable`. The code for this detector can be derived from the one of `FireDetector1` by not implementing input `disabled` and renaming input `smoke1` in `smoke2`. When `ControlUnit` and `FireDetector1` are composed, they synchronize on *fire* and *disable*. In the product, there are no locally incompatible states and the tool deduces that the two components can cooperate correctly in all environments. Note that the action *fire* survives in the composition both as an input and as an output, thus allowing `FireDetector1`  $\parallel$  `ControlUnit` to be composed with other fire detectors. The composition of `ControlUnit` and `Faulty_FireDetector1` goes less smoothly. When the composition receives a *disable* action, the control unit shuts down ( $s = 3$ ), while the faulty detector remains in operation. When the faulty detector senses smoke (input `smoke2`), it will emit *fire*: if the control unit has been disabled by the *disable* action, this causes an incompatibility. TICC diagnoses this incompatibility by synthesizing the following input restrictions:

- A restriction preventing the input *disable* if the faulty detector has detected smoke and is about to issue *fire*.
- A restriction preventing the input *smoke2* when `ControlUnit` is at  $s = 3$  (disabled).

Since the actions *disable* and *smoke2* should be acceptable at any time, the new input restrictions for these actions are a strong indication that the composition `ControlUnit || Faulty_FireDetector1` does not work properly. However, we conclude that the two components are compatible since they can work together in at least one environment.

The incompatibility in Example 1 is exposed by the following series of OCaml commands:

```
# open Ticc;;
# parse "fire.si";;
# let controlunit = mk_sym "ControlUnit";;
# let fire1 = mk_sym "FireDetector1";;
# let wfire2 = mk_sym "Wrong_FireDetector2";;
# print_input_restriction (compose controlunit wfire2) "disable";;
# print_input_restriction (compose controlunit wfire2) "smoke2";;
```

The function `print_input_restriction` is used to print the restricted version of the input invariant.

Note that the fire detection system illustrates the “many-to-one” communication model of sociable interfaces: several fire detectors can communicate with the control unit using the action *fire*. This is done by allowing fire detectors to receive the action *fire* also as an input. In previous models that have to distinguish between input and output names, this would lead to an incompatibility, and the only solution would be to give an individual name to each fire detector. This has the double disadvantage of bounding the number of detectors that can communicate with the control unit and increasing its number of transitions (and so the internal representation used in the tool).

Due to space limitations, the reader is referred to [11] for many other illustrations of the application and the utility of the composition operation in TICC.

## 7 More on the Symbolic Representation

As already announced in Definition 1, sets of states can be manipulated through TICC. A set of states can be defined in TICC via a formula specifying constraints on the values of the variables. TICC can parse such a formula, and construct a symbolic representation (an MDD) that enables it to manipulate the set. TICC can combine such sets with the usual Boolean operators through several functions. The tool also contains an implementation of the classical CTL operators [7]. Those operators can be used to verify properties of components via model checking. TICC implements CTL operators with a slightly modified (and more efficient) version of the basic symbolic algorithms given in [7]. Those algorithms are based on the computation of the fixpoint of predecessor operators similar to those defined in Section 5. When model checking CTL, TICC views the module as a closed system whose transitions are the input and output moves of the game it induces.

We describe now an example that illustrates the use of the symbolic representation and its application to CTL.

*Example 2.* Consider the fire detection system given in Section 5.1, and the script file in Figure 1. Line 10 builds the symbolic representation of a set  $\phi$  consisting of the states where `ControlUnit.s = 2`. Line 12 prints the set of states that satisfy the CTL formula  $\exists \diamond \phi$ , and line 14 prints the set of states that satisfy the CTL formula  $\forall \diamond \phi$ . Note that Line 8 of the script illustrates the use of the function `simulate`, which was informally introduced at the end of Section 5.1.

The previous example is very simple and only shows a small fragment of the functionalities provided by the tool. In [11], we give a more complex example that also involves the use of global variables and of a closure function. The closure function (`close` in TICC) allows the user to close a module with respect to the occurrence of input transitions. This can be used to say that the environment is no longer able to provide a certain input.

## 8 Refinement

The notion of refinement is introduced to capture the relation between an abstract model of a component and a more detailed one, or between a model expressing a specification and a model describing an implementation. In an input-enabled setting, refinement is usually defined as trace containment or a simulation [22]; this ensures that all output

```

1 open Ticc;;
2 parse "fire.si";;
3
4 let fire1 = mk_sym "FireDetector1";;
5 let controlunit = mk_sym "ControlUnit";;
6 let comp = compose fire1 controlunit;;
7
8 simulate comp "FireDetector1.s = 0 & ControlUnit.s = 0", 5, "r.html";;
9
10 let called_firemen = parse_stateset ("ControlUnit.s = 2");;
11 print_string "Can call the firemen:";;
12 print_stateset (ctl_e_f comp called_firemen);;
13 print_string "Always calls the firemen:";;
14 print_stateset (ctl_a_f comp called_firemen);;

```

**Fig. 1.** A script file illustrating CTL model-checking in TICC.

behaviors of the implementation are allowed by the specification. Unfortunately, such definition does not stand in an non input-enabled setting, since it does not forbid the implementation to make stronger assumptions on the environment than the specification does. To overcome the problem, de Alfaro et al. suggest a contravariant definition [12] which replaces simulation by *alternating simulation*. More precisely, an interface  $Q$  refines an interface  $P$  if each input transition of  $P$  can be simulated in  $Q$ , and each output transition of  $Q$  can be simulated in  $P$ .

In the rest of this section, we propose a definition of refinement for sociable interfaces. This definition is an extension of the one given in [9]. We consider  $M_1$  and  $M_2$ , two well-formed sociable interfaces, where  $M_i = (Act_i^G, Act_i^L, \mathcal{D}, V_i^G, V_i^L, V_i^H, W_i, \rho_i^{IL}, \rho_i^{IG}, \rho_i^O, \rho_i^L, \psi_i^I, \psi_i^O, I_i, V_i^{\text{all}} = V_i^L \cup V_i^L \text{ and } S_i = \llbracket V_i^{\text{all}} \rrbracket$ ). The sets  $Act_i^G, V_i^G, V_i^H$ , and  $W_i$  jointly define the *signature* of interface  $M_i$ .

**Definition 7 (Signature).** The signature  $Sign(M_i)$  of an interface  $M_i$  is the tuple  $(Act_i^G, V_i^G, V_i^H, W_i)$ .

The following result shows that signature equality preserves composability.

**Theorem 2.** Let  $N_1, N_2$ , and  $N_3$  be three interfaces, such that (1)  $Sign(N_1) = Sign(N_2)$ , and (2)  $N_2$  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.

We now define a notion of alternating simulation for sociable interfaces. In addition to what has been said above, the definition must take into account the fact that the environment of an interface cannot see the local transitions. Given a state  $s$  of an interface, we define  $\varepsilon$ -closure( $s$ ) to be the set of states that can be reached from  $s$  by applying zero or more local transitions.

**Definition 8 (Refinement Relation).** Assume that  $Sign(M_1) = Sign(M_2)$ . A relation  $\preceq \subseteq S_1 \times S_2$  is a *refinement relation* iff  $s \preceq t$  implies:

1.  $s[V_1^G] = t[V_1^G]$ ;
2. for all  $a \in Act_2^G$  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  $a \in Act_1^G$  and for all  $s' \in S_1$  such that  $\langle s, s' \rangle \models \widehat{\rho}_1^O(a)$  there exists  $t', t'' \in S_2$  such that  $t' \in \varepsilon\text{-closure}(t)$  and  $\langle t', t'' \rangle \models \widehat{\rho}_2^O(a)$  and  $s' \preceq t''$ .
4. for all  $a \in Act_1^L$  and for all  $s' \in S_1$  such that  $\langle s, s' \rangle \models \widehat{\rho}_1^L(a)$  there is a state  $t'$  in  $\varepsilon\text{-closure}(t)$  such that  $s' \preceq t'$ .

To gain some insight into Definition 8, consider that there is a refinement relation such that  $s \preceq t$  if  $M_1$  in state  $s$  can replace  $M_2$  in state  $t$  in every context, without the environment noticing any difference. Thus, first of all  $s$  and  $t$  must agree on the values of the global variables known to  $M_1$  (remember that  $V_1^G = V_2^G$ ). Then, each input that can be accepted by  $M_2$  from  $t$  must also be acceptable by  $M_1$  in  $s$ . Conversely, each output that can be emitted by  $M_1$  in  $s$  must also be emittable by  $M_2$  in  $t$ , or in another state  $t'$  that is invisibly reachable from  $t$ . Finally, all local transitions of  $M_1$  from  $s$  must correspond to zero or more local transitions of  $M_2$  from  $t$ . We derive from these definitions a concept of refinement for sociable interfaces.

**Definition 9 (Refinement).** We say that  $M_1$  *refines*  $M_2$  iff (i)  $Sign(M_1) = Sign(M_2)$ , and (ii) there is a refinement relation  $\preceq$  such that: for all  $t \models \psi_2^I \wedge \psi_2^O \wedge I_2$  there is  $s \models \psi_1^I \wedge \psi_1^O \wedge I_1$  such that  $s \preceq t$ .

**Theorem 3.** Let  $N_1, N_2$ , and  $N_3$  be three modules, such that (1)  $N_1$  refines  $N_2$ , and (2)  $N_2$  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.

A preliminary refinement operation has been implemented in the new version of the tool in a function called “refines” that takes two symbolic modules as arguments. The result is either `true` if the first module refines the seconde one, and `false` otherwise. The implementation has already been discussed in [9] for the initial model — the extension to the model of this paper follows similar principles.

## 9 Conclusion and Perspectives

This paper considers sociable interfaces — a concrete game-based model with rich communication primitives to facilitate the modeling of software and distributed systems.

The sociable interfaces model has lead to a new tool named TICC. Several tools implementing game-based models have already been built before TICC. As an example, we mention the tool CHIC that implements the synchronous, variable-based interface theory of [5] which is able to handle pushdown games, a feature that TICC does not have (yet). Another example is the Ptolemy toolset [19] that implements the model of [12].

With respect to these other tools, TICC has the advantage of being able to use rich communication primitives to model components in a very compact and natural way. The symbolic representation of TICC is also particularly attractive since it makes the tool very efficient and easily extensible. Both the tool and the sociable interfaces model are in constant evolution, and we are considering many improvements as well as several promising research directions.

One of our major concerns is the improvement of the existing functionalities. As an example, we plan to implement a new function that, given a symbolic set of states  $S$  and a CTL formula  $\phi$ , will check if all states in  $S$  satisfy  $\phi$  and will print a shortest counterexample trace if it is not the case. Such a functionality has been shown to be very useful for the design and the debugging of systems [23]. We are also considering the implementation of several functions that would give more feedback about a partial incompatibility between modules (see Example 1).

A new feature we are currently implementing consists in handling alternating-time temporal logic (ATL), a CTL-like logic designed for open systems, which has been proposed in [3]. The implementation is not straightforward since the algorithms of [3] first need to be framed into our model of game. As it is the case for the CTL logic, we also plan to consider counterexample traces.

Another promising research direction we are now investigating is a real-time extension of the Sociable Interface framework, along the lines of the *Timed Interfaces* of [14]. This is a large and complex endeavor, as the game-theoretic machinery of TICC will have to be replaced with one suited to real-time games [10]. An interesting application for the timed sociable interface model would be the scheduling of timed open systems. A timed version of the tool is desirable since it would constitute a platform for the implementation of various recent results on timed open systems (e.g., [18]).

As it is illustrated in [11], TICC is already able to deal with large systems. However, its efficiency is still limited by the existing algorithms that are used to manipulate the symbolic representation (see Section 5 and Section 7). In a future work, we plan to improve those algorithms by adapting recent works such as [6].

## Acknowledgement

We thank James Worrel and Damien Ernst for helpful comments on various drafts of this paper.

## References

1. S. Abramsky, D. R. Ghica, A. S. Murawski, and C.-H. Luke Ong. Applying game semantics to compositional software modeling and verification. In *Proc. 10th International Conference on Tools and techniques*, volume 2988 of *Lect. Notes in Comp. Sci.*, pages 421–435, Barcelona, Spain, 2004. Springer-Verlag.
2. B. Adler, L. de Alfaro, L. D. da Silva, M. Faella, A. Legay, V. Raman, and P. Roy. Ticc, a tool for interface compatibility and composition. In *Proceedings 18th International Conference on Computer Aided Verification (CAV)*, volume 4144 of *Lect. Notes in Comp. Sci.*, pages 59–62. Springer, 2006.

3. R. Alur, T.A. Henzinger, and O. Kupferman. Alternating-time temporal logic. In *Proc. 38th IEEE Symp. Found. of Comp. Sci.*, pages 100–109. IEEE Computer Society Press, 1997.
4. 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.
5. 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 Verification*, volume 2404 of *Lect. Notes in Comp. Sci.*, pages 414–427. Springer-Verlag, 2002.
6. G. Ciardo and A. J. Yu. Saturation-based symbolic reachability analysis using conjunctive and disjunctive partitioning. In *CHARME*, *Lect. Notes in Comp. Sci.*, pages 146–161. Springer-Verlag, 2005.
7. E.M. Clarke, O. Grumberg, and D.A. Peled. *Model Checking*. MIT Press, 1999.
8. L. de Alfaro. Game models for open systems. In *Proceedings of the International Symposium on Verification*, volume 2772 of *Lect. Notes in Comp. Sci.* Springer-Verlag, 2003.
9. L. de Alfaro, L. D. da Silva, M. Faella, A. Legay, P. Roy, and M. Sorea. Sociable interfaces. In *Proceedings of 5th International Workshop on Frontiers of Combining Systems*, volume 3717 of *Lect. Notes in Comp. Sci.*, pages 81–105. Springer, 2005.
10. L. de Alfaro, M. Faella, T.A. Henzinger, R. Majumdar, and M. Stoelinga. The element of surprise in timed games. In *CONCUR 03: Concurrency Theory. 14th Int. Conf.*, volume 2761 of *Lect. Notes in Comp. Sci.*, pages 144–158. Springer-Verlag, 2003.
11. L. de Alfaro, M. Faella, and A. Legay. An introduction to the tool ticc. Technical report, University of California Santa Cruz, 2006. Available at <http://luca.soe.ucsc.edu/Publications>.
12. 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.
13. L. de Alfaro, T.A. Henzinger, and O. Kupferman. Concurrent reachability games. Technical Report UCB/ERL M98/33, University of California at Berkeley, 1998.
14. L. de Alfaro, T.A. Henzinger, and M. Stoelinga. Timed interfaces. In *Proceedings of the Second International Workshop on Embedded Software (EMSOFT 2002)*, *Lect. Notes in Comp. Sci.*, pages 108–122. Springer-Verlag, 2002.
15. 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.
16. D.L. Dill. *Trace Theory for Automatic Hierarchical Verification of Speed-Independent Circuits*. MIT Press, 1988.
17. M. Faella and A. Legay. Some models and tools for open systems. Technical report, University of California at Santa Cruz, 2005. Proceedings of FIT05.
18. T. Henzinger and V. Prabhu. Timed alternating-time temporal logic. In *FORMATS06*, *Lect. Notes in Comp. Sci.*, Paris, France, 2006. Springer-Verlag. To appear.
19. E. A. Lee and Y. Xiong. A behavioral type system and its application in Ptolemy II. *Formal Aspect of Computing Journal*, 2003.
20. Xavier Leroy. Objective caml. <http://caml.inria.fr/ocaml/index.en.html>.
21. A. Lopes and J. Luiz Fiadeiro. Superposition: composition vs refinement of non-deterministic, action-based systems. *Formal Asp. Comput.*, 16(1):5–18, 2004.
22. N.A. Lynch. *Distributed Algorithms*. Morgan-Kaufmann, 1996.
23. K.L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, 1993.
24. 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.
25. S. Abramsky. Algorithmic game semantics and static analysis. In *SAS*, volume 3672, page 1. *Lect. Notes in Comp. Sci.*, 2005.
26. A. Srinivasan, T. Kam, S. Malik, and R. Brayton. Algorithms for discrete function manipulation. In *Proceedings International Conference CAD (ICCAD-91)*, 1990.
27. W. Thomas. On the synthesis of strategies in infinite games. In *Proc. of 12th Annual Symp. on Theor. Asp. of Comp. Sci.*, volume 900 of *Lect. Notes in Comp. Sci.*, pages 1–13. Springer-Verlag, 1995.