An Introduction to the Tool Ticc*

Luca de Alfaro[1], Marco Faella[2], and Axel Legay[3]

[1] Department of Computer Engineering, Universitity of California, Santa Cruz, USA
[2] Dipartimento di Scienze Fisiche, Università di Napoli "Federico II", Italy
[3] Department of Computer Science, University of Liège, Belgium

Technical report ucsc-crl-06-14
School of Engineering
University of California, Santa Cruz
Santa Cruz, CA 95064, USA

Abstract. This paper is a tutorial introduction to the sociable interface model of [12] and its underlying tool TICC [1]. The paper starts with a survey of the theory of interfaces and then introduces the sociable interface model that is a game-based model with rich communication primitives to facilitate the modeling of software and distributed systems. The model and its main features are then intensivelly discussed and illustrated using the tool TICC.

1 Introduction

The prevalent 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 by design teams. 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 others 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 a 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?") [21]. We describe here interface theories [13–15] a formal notion of component interfaces that use games to represent the interaction between the behavior originating within

* This research was supported in part by the NSF grants CCR-0234690 and CCR-0132780, by the ARP awards SC2005553 and SC20051123, and by a F.R.I.A Grant.

a component, and the behavior originating from the component's environment. Such an interface model is able to capture dynamic aspects of component interaction which makes it similar to a type system: indeed, it could be termed a "behavioral" type system for component interaction. In previous works, interface theories have been introduced for various aspects of component interaction: [13, 8, 7, 15, 12] consider the *protocol* dimension of interfaces ("What are the temporal ordering constraints on communication events between components?"), [16] considers the timing dimension of interfaces ("what are the real-time constraints on communication events between components?"), and [5] deals with constraints on the resource usage of the component.

In this paper, we focus on the *sociable interfaces* model introduced in [12], and on the corresponding tool called TICC [1]. We present the underlying ideas of the model, and show how it can be used to capture the protocol dimension between components. All the concepts are intensively illustrated with TICC for which this paper constitutes an introduction.

Two tools for interface theories predate TICC. The asynchronous, action-based interface theories of [13] are implemented as part of the Ptolemy toolset [19]. The tool CHIC [6] implements synchronous, variable-based interface theories modeled after [14]. Our goal in developing TICC is to provide an asynchronous model where components have rich communication primitives that facilitate the concise, natural modeling of software and distributed systems. In TICC, components are modeled both via variables (to describe state) and actions (to describe synchronization); its communication primitives enable the modeling of complex communication schemes. The implementation of TICC relies on symbolic methods, yielding efficient algorithms for component and system analysis.

2 Interface Theories

Before going to the details of the sociable interfaces model, we first summarize and illustrate the basic features of Interface theories. The reader is referred to [17, 10, 18, 12] for more details.

Interface Specification and Well-formedness

An interface specifies how a component interacts with its environment. It describes the input assumptions that the component makes on the environment and the output guarantees it provides. Interfaces capture the I/O behavior of a component by an automaton whose syntax is similar to the I/O automata of [21]. In the context of software design, inputs are used to model procedures or methods that can be called, and the receiving end of communication channels, as well as the return locations from such a calls. Outputs are used to model procedure or method calls, message transmissions, the act of returning after a call or method terminates, and exceptions that arise during method execution. Unlike traditional models of open systems, among which I/O automata, that at every state must be receptive to every possible input event, in interfaces it is possible that inputs are illegal (cannot be accepted) at some states. Thus, an interface describes

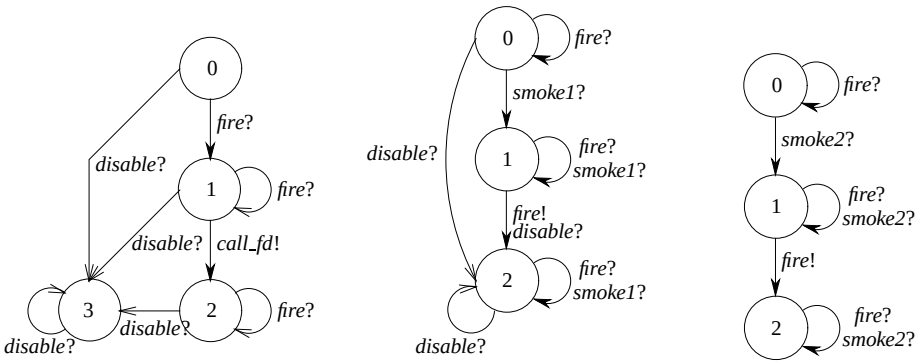
the behavior of a component only with respect to some environments. In this way, environment restrictions can be used to encode restrictions on the order of method calls, and on the types of return values and exceptions. This is how interfaces capture the protocol dimension of components. Another advantage of making explicit assumptions about the environment is that it gives rise to an optimistic compatibility test when interface are composed: two interfaces are compatible if there exists at least one environment in which they can work together. Finally, from a practical point of view, the ability to forbid inputs removes the need to specify "what happens" when taking an undesirable input. Such a specification has been pointed to as one of the main drawbacks of input-enabled approaches. Since we can make input assumptions, we have to ensure that the interface is *well-formed*, i.e. that there exists at least one environment that satisfies its input assumptions.

Interfaces as Games

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 if the Input player has a winning strategy in the game, which means that the environment can meet all input assumptions. Games provides a model for multiple independent sources of nondeterminism and keep the distinction between inputs and outputs. Hence, even if the syntax of interfaces is close to the one of I/O automata, they differ in the way that the operations on the models are defined. In this paper, we will mainly mainly focus on the operation of composition between two or more interfaces.

Interface Composition and Compatibility

The game-like nature of interfaces becomes apparent when we consider the operation of composition. In their original formulation, 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 expecting the environment to 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. Component 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. The optimistic approach supports incremental design: the compatibility of two components can be checked without specifying interfaces for all components of the system, i.e. without closing the system.



(a) C: Control Unit

(b) D_1: Smoke Detector

(c) D_2: Faulty Smoke Detector

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

Incremental designs also ensure that compatible components can be put together in any order.

An Example

We illustrate the previous concepts with the help of a simple example: a fire detection system. The system is composed of a control unit and several smoke detectors. The interfaces for this example are reported in Figure 1: D_1 is one of the smoke detectors (there could be more), and C is the control unit.

When a detector senses smoke (input event *smoke*?), it reports it by emitting the output event *fire*!. When the control unit receives the input event *fire*? from any of the detectors, it issues a call for the fire department (output event *call_fd*!). Additionally, an input event *disable*? disables both the control unit and the detectors, so that the smoke sensors can be tested without triggering an alarm. We also suppose the existence of faulty smoke detectors, i.e., smoke detectors that ignore the *disable* message. The interface for a faulty smoke detector is presented in Figure 1(c).

A particularity in the design is that some (but not all) states are input-enabled. As an example, state 1 of C is still receptive to the input event *fire*? after receiving the smoke alarm. This is because detectors are independent and there is thus no reason for one detector to be forbidden to send output even *fire*! if this has already been done by some other detector. Another example is state 2 of D_1 which is receptive to the input events *fire*? and $smoke_1$?. Note that the possibility of having the same name for input and output events is proper of sociable interfaces model and not allowed in other interface models presented in [13, 8, 7, 15], or even transition based models (such as I/O automata). This

illustrates the multiple ways of communicating that are allowed by the model (see [12] for a discussion). Note that C, D_1, and D_2 are well-formed[4].

It is easy to see that all the states in the composition between the control unit C and the fire detector D_1 will be compatible if the two interfaces communicate via the event *fire*. As an example, if C is in state 1 and D_1 in state 2, then the output event *fire*! emitted by D_1 can be caught by the input event *fire*? of C. The output event *call_fd*! emitted by C does not need to be caught by D_1 since the two interfaces do not synchronize on this action.

However, the composition between C and D_2 goes less smoothly. When the composition receives the input event *disable*?, the control unit shuts down ($s = 3$) and makes the assumption that the environment cannot emit any output, while the faulty detector remains in operation. When the faulty detector senses smoke (input event *smoke*2?), it emits the output event *fire*!: if the control unit has been disabled, this causes a local incompatibility in state $(3,1)$. Hence, a winning strategy for player Input to stay away from locally incompatible states can be realized by the following input restrictions:

- A restriction preventing the input event *disable*? if the faulty detector is in state $s = 1$, that is, it has detected smoke and is about to issue the output event *fire*!.
- A restriction preventing the input event *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 between C and D_2 does not work properly in all environments. However, if we consider an environment that never issues *disable*, then the two interfaces can work together in a proper way.

3 The Sociable Interfaces Model

This section sketches the main elements of a sociable interface and the game it induces. The reader is referred to [12] for more details. A sociable interface M is composed of the following elements.

- A set of *global actions* Act^G and a set of *local actions* Act^L.
- A set of *variables* V^{all} which is partitioned into a set of *local variables* V^L and a set of *global variables* V^G, with $V^G \cap V^L = \emptyset$. Local variables are used to describe the internal states of the interface, while global variables are used to describe the global state of the system. Among the set of global variables, we distinguish between *history* and *history-free* variables. This distinction, which will be discussed in Section 5, allows us to limit the number of actions an interface should include. The set of history variables is denoted by V^H.
- A set of input and output *transitions*. Each global action $a \in Act^G$ is associated to an input and an output transition that are respectively denoted by $\rho^I(a)$ and $\rho^O(a)$. An output transition specifies how variables are updated when the interface emits the action. An input transition is the conjunction of two parts: *(1)* an input global

[4] In general, checking well-formedness requires solving a safety game [12]

transition $\rho^{IG}(a)$ that specifies constraints on how other interfaces can update the global variables when emitting a, and *(2)* an input local transition $\rho^{IL}(a)$ that can update the local variables of the interface when other interfaces emit a. The reason to split the input transitions in two parts will be discussed in Sections 4 and 5.
- A set of *local transitions*. Each local action $a \in Act^L$ is associated with a transition $\rho^I(a)$, which can modify the value of local variables. Local transitions cannot be synchronized with transitions of other interfaces. Notice that local transitions were not present in the original model of [12].
- An input and an output invariant, respectively denoted by ψ^I and ψ^O. Invariants are sets of states that are used to constrain the input and output transitions of the interface. Precisely, input transitions must maintain the input invariant true, and output transitions must maintain the output invariant true.
- An initial condition I that describes the initial constraints on the set of local variables of the interface.

For an interface M, we say that a *state* of M is a value assignment to the variables in V^{all}.
Note that in TICC, the term "sociable interface" is replaced by "module", but a module is no more than the description of a sociable interface in the input language of the tool.

Example 1. The Control Unit C of the fire detection system described in the previous section is a sociable interface with 3 actions: *fire*, *disable*, *call_fd*. Its internal state can be encoded with a local variable s, so that a state of C assigns a value between 0 and 3 to s. The action *fire* has input transitions from states $s = 0, s = 1, s = 2$, but not from $s = 3$. The action has no output transition meaning that the interface cannot emit the action *fire*. The action *call_fd* has an output transition from $s = 1$ to $s = 2$, but no input transition. Hence, the interface makes the assumption that the environment can never issue *call_fd*. Notice that we can follow the same reasoning for the (faulty) smoke detectors.

The Game Underlying the Model

As mentioned in the introduction, a sociable interface induces a turn-based game between the Input and the Output players. The definitions of the moves, outcomes, and strategies of this game have been described in [12]. In this paper, the reader only needs to know how moves are defined.

The moves of the Input and Output players are those induced by the input and the output transitions (the game model supposes that both the input and the output transitions are conjoined with their corresponding invariants). In addition, each player owns a stuttering move to ensure that the runs of the game are infinite.

The definition of the stuttering move is straightforward for the Output player: this is the identity transition. For the Input player the definition is slightly different: the stuttering move is an additional transition that can modify the value of global history free variables. The stuttering move of the Input player is often referenced to as the *environment transition*; it is automatically added by TICC when specifying a sociable interface.

Well-Formedness

Given a sociable interface M, it is possible to compute the set of states S_I (resp. S_O) from which the Input (resp. Output) player has a strategy to always stay in the set of states that satisfy the input (resp. output) invariant, whatever the Output (resp. Input) player does. A sociable-interface is well-formed if each reachable state s of M belongs to $S_I \cap S_O$, and moreover $\psi^I = S_I$ and $\psi^O = S_O$; see [12] for a detailed explanation.

The tool TICC automatically ensures that modules are well-formed before allowing the user to manipulate them. To this end, TICC may add extra conditions to the initial condition and the input/output invariants that are defined by the user. Hence, the user does not need to take care of the notion of well-formedness, and we will not elaborate on it in the rest of the paper.

The Tool TICC

TICC is a tool that 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. TICC is implemented as a set of functions that extend the capabilities of the OCaml [20] command-line. The tool is released under the GPL. The code of TICC is freely available and can be downloaded from *http://dvlab.cse.ucsc.edu/dvlab/Ticc*. This web site is a Wiki that also contains the documentation for the tool, as well as several examples including those that will be presented in this paper. Internally, TICC relies on a representation of modules which is based the MDD/BDD Glue and Cudd packages [22]. The source files of the tool are organized as follows:

1. The root contains files with the basic information about the tool. There is a `README` file that describes the files of the root.
2. The directory `examples` contains a series of examples and a tutorial. Again, there is a `README` file that can be consulted for more information.
3. The directory `src` contains the code itself; it is composed of three sub directories: `glu-2.0`, `mlglu`, and `ticc`. Directories `glu-2.0`, and `mlglu` contain the code needed to adapt the MDD/BDD Glu package to work with tcc. A directory `doc` contains automatically-generated documentation for the tool. In particular, the file `doc/api/Ticc.html` (automatically generated from `src/ticc/ticc.mli`) documents all the commands available to the user.

4 Starting with TICC

This section is an introduction to the use of TICC. It presents very simple examples, to illustrate the process of entering a program, and running the tool.

To use TICC, first ensure that the executable file "ticc" is in your path. Then, invoke it in interactive mode simply by typing:

```
ticc
```

The result of this operation is an Ocaml prompt[5] 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*. Most of them will be described in the rest of this paper.

The next operation is to provide TICC with a TICC program. TICC programs are entered in files with the extension .si that stands for **S**ociable **I**nterface. The syntax of TICC programs will be presented in the following sections. Program files are parsed with the command

```
parse "MyTiccProgram.si";;
```

The `parse` function reads in a .si file describing modules and 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. Parsing multiple files is allowed and viewed as an incremental process: new declarations are added to the existing ones. This implies that one cannot declare two modules with the same name in different files, and that one only needs to declare global variables once. After parsing at least one TICC program, one can perform operations on and between modules of the program.

Notice 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. Figure 2(b) provides an example of the content of a script file whose name is `example.in`. At this point, the reader should be able to interpret lines 1 and 2. Lines 3 and 4 will be explained later. One can invoke TICC to execute the script file with the following command from the shell prompt:

```
ticc example.in
```

We dedicate the rest of this section to TICC programs that illustrate some of the main features of the input language of the tool. Operations on and between modules will be described in the next sections.

4.1 Getting to Know TICC Programs

As a first example, we consider the translation of the fire detection system to a TICC program. The file for the corresponding TICC program is given in Figure 2(a) and is named `detector.si`.

The program consists in the declaration of three modules: Module `ControlUnit`, `FireDetector1`, and `Faulty_FireDetector2` respectively correspond to interfaces C, D_1, and D_2 of Figure 1. Let us consider module `ControlUnit`. This module shows some of the very basic elements of a TICC module. It contains:

– Local variable declarations. The module declares a variable `s` whose value is an integer between 0 and 3. TICC supports Boolean and integer range variables.

[5] Remember that TICC is implemented as a set of functions that extend the capabilities of Ocaml.

```
1  module ControlUnit:
2      var s: [0..3] // 0=waiting, 1=alarm raised, 2=fd called, 3=disabled
3
4      input fire:    { local: s = 0 | s = 1 ==> s' := 1
5                       else  s = 2        ==>        }
6      input disable: { local: true ==> s' := 3 }
7      output call_fd: { s = 1 ==> s' = 2 }
8  endmodule
9
10 module FireDetector1:
11     var s: [0..2] // 0=idle, 1=smoke detected, 2=inactive
12
13     input smoke1: { local: s = 0 | s = 1 ==> s' := 1
14                     else   s = 2 ==>      } // do nothing if inactive
15     output fire:  { s = 1       ==> s' = 2 }
16     input fire:   { } // accepts (and ignores) fire inputs
17     input disable: { local: true ==> s' := 2 }
18 endmodule
19
20 module Faulty_FireDetector2:
21     var s: [0..2] // 0=idle, 1=smoke detected, 2=inactive
22
23     input smoke2: { local: s = 0 | s = 1 ==> s' := 1
24                     else   s = 2 ==>      } // do nothing if inactive
25     output fire:  { s = 1       ==> s' = 2 }
26     input fire:   { } // accepts (and ignores) fire inputs
27     // does not listen to disable action
28 endmodule
```

(a) TICC modeling of a fire detector system: `detector.si`.

```
1  open Ticc;;
2  parse "detector.si";;
3  let controlunit = mk_sym "ControlUnit";;
4  let faulty = mk_sym "Faulty_FireDetector2";;
```

(b) A script file that parses `detector.si`.

Fig. 2.

```
1  (* open the functionnalities of the tool *)
2  open Ticc;;
3
4  (* parse the file in where modules are described *)
5  parse "detector.si";;
6
7  (* create the symbolic representations for the three modules declared
       in fire-detector.si *)
8  let fire1 = mk_sym "FireDetector1";;
9  let faulty = mk_sym "Faulty_FireDetector2";;
10 let controlunit = mk_sym "ControlUnit";;
11
12 (* print the input and output invariants of symbolic module fire1 *)
13 print_symmod_iinv fire1;;
14 print_symmod_oinv fire1;;
15
16 (* print the transition rule corresponding to action "fire" in module
       fire1 *)
17 print_symmod_rules fire1 "fire" ;;
18
19 (* print the entire symbolic module fire1 *)
20 print_symmod fire1;;
```

Fig. 3. The TICC script `detector.in` for the fire detector system.

– Input and output transitions. 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 `call_fd` can be taken only when s has value 1; the transition leads to a state where $s = 2$. The declaration of the local part of an input starts with the keyword `local` (and so the global starts with `global`). This declaration has a particular structure, to ensure that the local part of the rule is deterministic (see next section for clarification).

The code of `detector.si` presents other features that will be extensively discussed in other examples.

An example of a script file for the fire detection system is given in Figure 3. The name of this file is `detector.in`. Let us briefly describe what happens when executing `ticc detector.in` from the shell.

Code between lines 1 and 5 has already been described earlier: we open the tool and parse a TICC program specified in a file called fire-detector.si. At this point, TICC contains an enumerative representation of the modules and the global variables that have been declared.

The command `mk_sym` used in lines 8, 9, and 10 converts the enumerative representation of modules into a symbolic representation based on MDDs [23]. An MDD is similar to a BDD [4], extended to work on integer ranged variables instead of Boolean

ones. Given a constraint on a set of integer ranged variables, an MDD is a representation of all the values of the variables that satisfy the constraints.

The initial condition and the input/output invariants of a module are sets of constraints on its variables; they can thus be represented with MDDs. Since transition relations express constraints between the values of the variables before and after the transitions have been applied, they can also be represented with MDDs. The symbolic representation is in general more compact and efficient than an enumerative one; TICC operations can be easily implemented symbolically, as explained in [12].

The rest of the file `detector.si` illustrates some of the printout functions available in TICC. As an example, in lines 13 and 14 the user asks TICC to print out the input and output invariants of the symbolic module `fire1`. In this example, both invariants have value `true`. In line 17 the user asks TICC to print the transition rule corresponding to action `fire` of module `FireDetector1`. The printout produced by this command is:

```
PRINTING the rule(s) for the action fire of
SYMBOLIC MODULE: FireDetector1.
[input part]:
modified vars:
{ }
[input global part]:
(1)
[input local part]:
(1)
[output part]:
Owned by module FireDetector1
modified vars:
{ FireDetector1.s }
(
 (FireDetector1.s = 1)(
   (FireDetector1.s' = 2)) )
```

When performing a printout, TICC describes the input and output transitions corresponding to the action, as well as the variables that are involved. Notice that a condition which is `true` is denoted by TICC as "`(1)`". For more printout functions, consult the documentation file *ticc/doc/api/Ticc.html*.

4.2 A More Elaborate Example

We now present a more elaborate example of TICC module, that makes use of most features of the input language. An Anti-blocking System (ABS) is an automotive component that tries to prevent wheel slippage by modulating the braking force. In Figure 4, we present a model of an abstract ABS, comprising two modules. Module `ABS_controller` is intended to be periodically invoked by the environment using the action `tick`. When it receives that action, the module moves to the internal state `state`=1 and sets the global variable `abs_on` to `true`. Then, it checks the current acceleration of the vehicle against the current pressure of the user on the brake pedal. If the module establishes that the situation requires ABS intervention, it emits action `do_it`, otherwise it goes back to internal state `state`=0 via the action `reset`.

```
1  var b_pedal, b_force: [0..5]
2  var accel: [0..10]
3  var abs_on: bool
4
5
6  module ABS_controller:
7     var state: [0..2]
8
9     stateless accel, b_pedal
10
11    initial: state = 0
12
13    input update_b_force: { global: abs_on ==> b_force' = b_force }
14    input tick: { global: abs_on ==> b_force' = b_force
15                  local: state = 0 ==> state' := 1
16                    else true ==> }
17    output do_it: {
18       state = 1 & (b_pedal > 0 & accel > 4) ==> state' = 2 & abs_on'
19    }
20    output reset: {
21       state = 1 & (b_pedal = 0 | accel <= 4) ==> state' = 0 & ~abs_on'
22    }
23    input done: { local: state = 2 ==> state' := 0 }
24
25 endmodule
26
27
28 module ABS_actuator:
29    var turn, state: bool
30
31    stateless b_pedal, b_force
32
33    initial: turn = false & state = false
34
35    oinv: true
36    iinv: true
37
38    input do_it: { local: ~state ==> state' := true }
39    output done: {
40       state & turn ==> b_force' = b_pedal & ~turn' & ~state';
41       state & ~turn ==> b_force' = 0   &  turn' & ~state'
42    }
43 endmodule
```

Fig. 4. TICC modeling of an Anti-blocking System.

Module ABS_actuator, instead, accepts an input signal do_it. At that time, it moves to a different internal state characterized by state=true. When state=true, the module controls the brakes according to a simplified anti-blocking algorithm.

In the following, let M be the sociable interface corresponding to module ABS_actuator.

Global variables. Global variables are declared outside modules. As we will see, multiple modules can read and modify the value of global variables.

In our case, the system comprises four global variables: abs_on indicates whether the ABS is currently controlling the brakes, b_pedal is the amount of pressure that the driver is currently applying on the brake pedal, b_force is the amount of pressure that the brake pads are currently applying to the brake rotors, and accel is the current acceleration of the vehicle. Since TICC does not support negative ranges, we assume that values of accel smaller than 4 represent negative accelerations.

In TICC, the set of global variables used by a module is automatically built by collecting all global variables that are mentioned in any transition rule. Thus, as far as module ABS_actuator is concerned, we obtain $V_M^G = \{\texttt{b\_pedal}, \texttt{b\_force}\}$.

History-free variables. By default, a module remembers the value of its global variables, and expects to know all actions that can modify them. More precisely, by default, global variables in a module are *history* variables. The module assumes that, unless some input or output action modifies their value, these global history variables retain their value through time. To enable reasoning about their value, if a global variable is a history variable in a module M, all the actions that can modify this variable must be known to M (declared as input).

This requirement can potentially require a module to possess very many input actions. There are two solutions to this problem. One, *wildcard actions*, will be described later. The other solution consists in declaring some variables to be *history-free*. In this case, the module does not track their value, and does not need to know (declare) all actions that modify their value.

In the case of module ABS_actuator, both b_pedal and b_force are declared to be history-free. b_pedal is naturally history-free, since we can make no assumptions on how the driver is going to use the brake pedal. b_force is also left history-free, as we assume that the actuator does not care if other modules change its value. Since no other global variable is mentioned by the module, we obtain $V_M^H = \emptyset$.

Local variables. Local variables are declared inside a module, using the same syntax of global ones. A local variable is only visible in the module it is declared in.

Module ABS_actuator declares two local variables of type bool, so that $V_M^L = \{\texttt{state}, \texttt{turn}\}$. state is true when the module is ready to emit its output action. turn is used to implement the following simplified anti-blocking algorithm: when turn is true, the actuator lets the driver decide the amount of braking, when turn is false, the actuator sets the braking force to zero.

Actions. In TICC, actions are not specifically declared. One can directly declare a transition rule and label it with a new or pre-existing action name. The tool collects all the

```
var x, y: [0..10]

module Test:
   oinv: x + y <= 15
   output a: { true ==> x' = x + 1 }
endmodule
```

Fig. 5. A module with a non-trivial output invariant.

actions used by a module in a set of module actions.
For module `ABS_actuator`, we have $Act_M^G = \{\texttt{do\_it}, \texttt{done}\}$ and $Act_M^L = \emptyset$.

Initial condition. A module can declare its initial condition using the keyword `initial`. The initial condition is expressed by a Boolean expression over the set of local variables.

In our case, module `ABS_actuator` starts with `turn` and `state` equals to `false`.

Invariants. An invariant is a condition over the state space of a module, that is constantly satisfied. Following the input/output duality which is proper of interfaces, modules can have two invariants: an input invariant and an output invariant. The output invariant defines a set of states that will not be left by any local or output transition. In practice, each local or output transition rule is implicitly conjoined with the output invariant of the module. Dually, a module assumes that its environment does not violate its input invariant. In practice, all input transition rules are implicitly conjoined with the input invariant of the module. Note that, since modules are well-formed, the Input (resp. Output) player can ensure that the input (resp. output) invariant is never left. This indicates that no output transition leads from a state satisfying both invariants to a state satisfying the output, but not the input, invariant. Symmetrically, no input transition can lead from a state satisfying both invariants to a state satisfying the input, but not the output, invariant.

The invariants of `ABS_actuator` are both equals to `true`. In fact, specifying a `true` invariant is equivalent to specifying no invariant at all, as done by module `ABS_controller`. Invariants are useful to express certain relationships between variables. As instance, consider the example in Figure 5, comprising a module `Test`, together with two global variables.

The output invariant expresses the property that this module will always enforce that the sum of `x` and `y` is at most 15. This implies that module `Test` will not emit action `a` when the current sum of `x` and `y` is at least 15. As we will see later, the main use of invariants is in composition: input invariants will be used to express the constraints on the environment that guarantee the compatibility of the modules being composed.

Transition rules. TICC supports three types of transition: input, output and local transitions. Output transitions are the ones that users are most likely to be familiar with. They describe a possible behavior of the module, consisting in emitting an action, while possibly changing the value of global and local variables. Local transitions can be thought

of as a special type of output transition, where the module is only allowed to update its local variables. Moreover, local transitions are invisible to other modules, so that the name of the action labeling a local transition is irrelevant. They can be declared using the syntax:

```
local a: { guard ==> command }
```

Module `ABS_actuator` can only emit one output action, called `done`. As previously said, the corresponding transition rule is expressed by a sequence of guarded commands. In this case, the first guarded command (line 17) states that if both `state` and `turn` are true, action `done` can be performed. As a consequence, the next value of `b_force` will be equal to the current value of `b_pedal`, and both `state` and `turn` will have value false. The second guarded command (line 18) states that the transition can also be taken if state is true and turn is false. In this case, the next value of the global variable `b_force` will be zero, while the local variables `turn` and `state` will have value true and false, respectively. In this case, the two guards are mutually exclusive. In general, more than one guard can be true at a given time: at run-time, any of those guards can be selected nondeterministically.

Notice that action `done` occurs only as output in `ABS_actuator`. This implies that the module does not accept it as input.

One feature of TICC guarded commands that might surprise at first is that the distinction between guard and command is purely conventional. A guard and its corresponding command are internally conjoined, so that

```
guard ==> command
```

is always equivalent to:

```
true ==> guard & command
```

This holds for output rules, local rules, and the global section of input rules. The local section of input rules follows a different syntax, as explained later in this section.

For instance, consider again module `Test` in Figure 5. The transition rule corresponding to action "`a`" seems to state that module `Test` can always emit "`a`", whose effect will be to increase the value of `x`. However, according to the principle we just stated, the action cannot in fact be emitted when `x=10`.

Input transition rules are split in two sections. The *global* section describes assumptions about how other modules can change the value of global variables when emitting certain outputs. The *local* section describes how this module reacts when receiving a certain action. The reaction of the module to an input has two important restrictions: *(i)* it can only update local variables, and *(ii)* it must do so in a *deterministic* fashion. These restrictions are due to the theoretical assumption that each step is driven by the module carrying out the output action. In turn, this ensures that the semantics of the model is a turn-based game rather than a concurrent one. As a consequence, we have the following special syntax for the local part of input rules:

```
    guard1 ==> var11' := expr11, var12' := expr12, ...
else guard2 ==> var22' := expr21, var22' := expr22, ...
...
```

To ensure determinism, commands can only include assignments to local variables. Moreover, the `else` keyword is inserted to remind the user that in this context guarded commands will be evaluated in the order in which they are written, (i.e., `guard2` is evaluated only if `guard1` is false, and so on).

The only input action that module `ABS_actuator` can accept is called `do_it`. The corresponding transition rule has no global section, meaning that the module makes no assumptions on the current and next value of global variables when `do_it` is received. The local section states that, when `state` is false and `abs_on` is true, the next value of `state` will be `true`. We may wonder what happens when the conditions set by the guard fail (i.e., `state` is true or `abs_on` is false). The answer is that the condition expressed by the guard becomes an input assumption and as such it *migrates* to the global part of the rule, as witnessed by a printout of the module. In other words, the input rule corresponding to action `do_it` is equivalent to the following:

```
input do_it: {
   global: ~state ==> true
   local: true ==> state' := true
}
```

4.3 Arithmetic in TICC

TICC allows the declaration of Boolean and integer range variables. Both of those declaration have previously been illustrated. However, due to the bounded size of the variables, dealing with integer range variables implies some implementation choices that are worth summarizing.

From the previous section, we learned that integer range variables allow to build numerical expressions, while Boolean variables allow to build Boolean expressions. The two types of expressions are combined in guarded commands with the classical Boolean and numerical comparison operators. The question arises of how to interpret the arithmetical operators $+$ and $-$ on a finite range type. A common choice is to implement modulo arithmetic: for instance,if x and y have range $[0..\ m-1]$, then the expression $x+y$ is evaluated to $x+y \mod m$. This is the choice followed, for instance, in Mocha [3, 11]. There are two drawbacks in following this choice. The first is that comparisons behave in a counterintuitive way, making the system prone to modeling errors. For instance, the two comparisons $x+1 \geq y$ and $x \geq y-1$ are *not* equivalent: the first returns an unexpected result with $x = 3$, the second when $y = 0$. The second drawback is that it is difficult to come up with consistent and intuitive typing rules for expressions including variables with different ranges; for instance, it is not clear how to evaluate $x+y+z = w$ if all of x, y, z, and w have different ranges. Indeed, the tool Mocha avoided this problem by forcing expressions to consist of one range type only, which is a rather restrictive requirement.

In TICC, we follow a different choice, based on the following two principles:

1. Numerical expressions are always evaluated in a range that is large enough so that no roll-over, or overflow, occurs.
2. Negative numbers are not considered.

Let us illustrate the consequences of these principles. Consider the expression:

```
x' = y + z - 3
```

and assume that the ranges are as follows:

```
var x: [0..4]
var y: [0..5]
var z: [0..5]
```

The design decisions imply that:

1. The sum of y and z is evaluated in a temporary range type that is at least $[0..10]$, so that no overflow can occur.
2. If the result of the expression is negative, it is considered different from the result of any other expression, and in particular x', so that the overall expression will be false.

The expression is thus evaluated as follows:

- If x is 4, y is 4, and z is 3, then the expression $x' = y + z - 3$ will be true, as expected. In fact, $4 + 3$ will give 7, and $7 - 3 = 4$: no overflow occurs.
- If x is 1, y is 4, and z is 5, the expression is false, as $4 + 5 - 3 = 6$, which is different from 1. Note in particular that roll-over does not occur: even though $6 \mod 5 = 1$, the expression on the right hand side is considered to have value 6, not 1, in spite of the left hand side having range $[0..4]$.
- If y is 1, and z is 1, the expression will be false, since the right hand side gives rise to a negative number.

The evaluation of an expression proceeds by evaluating sub-expressions and by combining the obtained results. In general, one could suppose that if a sub-expression is evaluated to false, then the entire expression is evaluated to false. As an example, consider the following expression:

```
x' = y - z + 3
```

If x' is 2, y is 2, and z is 3, then the expression would yield value false because $y - z$ represents a negative number. However, we have that $2 = 2 - 3 + 3$, meaning that the evaluation of the whole expression is true! To mitigate (but not eliminate) this, after parsing, TICC tries to reorder the expressions, so that whenever possible, negative results are avoided. For instance, the above expression would be internally transformed into the following expression:

```
x' = y + 3 - z
```

so that a negative result would occur only if the total result is negative. TICC can do basic expression simplification, and it reorders the terms of a sum so that positive terms occur before negative terms. A good way for the user to know if reordering occurred is to print the syntactic representation of a module after parsing it.

```
open Ticc;;

parse "fire-detector-disable.si" ;;

let controlunit = mk_sym "ControlUnit";;
let fire1 = mk_sym "FireDetector1";;
let wfire2 = mk_sym "Faulty_FireDetector2";;

let c = compose fire1 controlunit;;
let d = compose wfire2 controlunit;;

print_symmod c;;
print_symmod d;;

print_input_restriction c "disable";;
print_input_restriction d "disable";;

print_input_restriction c "smoke1";;
print_input_restriction d "smoke2";;
```

Fig. 6. A script file illustrating the composition of the modules for the fire detector example of Figure 2(a).

5 Composing Sociable Interfaces in TICC

In TICC, the main operation on modules is *composition.* Composition synchronizes two modules on their shared actions, and returns a new module, representing the joint behavior of the two original modules, along with the environment assumptions required to guarantee the correct functioning of the original modules. While composing modules, TICC checks their *composability* and *compatibility:*

- *Composability* is a condition involving the sets of variables and actions of a module, and that can be checked statically, and extremely efficiently. Essentially, two modules are composable if it makes sense to consider the effect of their communication.
- *Compatibility* is a condition about the behavior of the modules. Two modules are compatible if there is some environment in which they can work jointly together, with all their input assumptions being satisfied. Checking compatibility requires solving a game between the Input and Output player; the solution of the game yields the input assumptions for the composition of the two modules.

The TICC command `compose` checks composability and compatibility of two modules, and if both tests are positive, computes a symbolic module corresponding to their composition. If incompatibilities arise, TICC can provide diagnostic information to detect the reason.

Example 2. The script file given in Figure 6 illustrates the composition operation for the fire detector example mentioned in Section 2 and Figure 2(a).

In the sociable interface model, and thus in TICC, the composition is done in four steps. First, one checks that the modules can be composed (see Section 5.1). If the modules are composable, then the next step is to build the product between them (see Section 5.2). At this point, the product can contains bad states, i.e. states that exhibit a local incompatibility (see Section 5.3). The last step of the composition consists in synthesizing a strategy for the Input player to stay away from the set of bad states whatever the Output player does (see Section 5.4).

This section describes how those four steps are conducted in TICC. More information about the theory behind the operations can be found in [12].

We remark that the composition of two modules in TICC only works on their symbolic representation. In what follows, we consider two symbolic modules M_1 and M_2 where $M_i = (Act_i^G, Act_i^L, V_i^G, V_i^L, V_i^H, \rho_i^I, \rho_i^O, \rho_i^L, \psi_i^I, \psi_i^O)$, and we implicitly refer to their corresponding sociable interfaces.

5.1 The Composability Condition

To facilitate composition, TICC ensures that modules have distinct local actions and local variables by automatically renaming local variables and local actions: a local variable x of module M is renamed to $M.x$ upon parsing the module M.

We say that the two modules M_1 and M_2 are *composable* if they satisfy the following non-interference condition: if an action $a \in Act_1^G$ (respectively Act_2^G) of module M_1 (resp. M_2) can modify a history variable of module M_2 (resp. M_1), then $a \in Act_2^G$ (resp. Act_1^G).

Since output transitions are the only ones that can modify the value of a global variable[6], the condition boils down to checking that if module M_1 has an output transition for action a that modifies[7] a history global variable of module M_2, then module M_2 must have an input transition for action a.

The non-interference condition is the main motivation for distinguishing between 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. Wildcard actions, as described later, is another method.

Example 3. Consider the composition of the modules in the Anti-blocking System (ABS) described in Section 4.2. The global variable `b_force` is a history variable for module `ABS_controller`. Since module `ABS_actuator` has an output transition for action `done` that modifies this variable, module `ABS_controller` *must* accept `done` as input. In this case, the input transition of action `done` states that module `ABS_controller` agrees on all modifications that could be done to the variable.

Another consequence of the non-interference condition is the following. Denote *ABS* the module obtained by composing the two ABS modules. If another module wants

[6] Input transitions only make assumptions on those values.

[7] Where "modifies" means that the the variable appears primed in the command of the output transition.

to modify variable `b_force` and be composed with *ABS*, it is forced to do so using one of the remaining inputs of *ABS*, namely `tick` and `update_b_force`. Both those input transitions impose the condition that if `abs_on` is true, the value of `b_force` is not modified. Thus, the non-interference condition allows modules to effectively control a global variable, when needed.

5.2 The Product

The product describes how elements of M_1 and M_2 are combined to give rise to a new module M_{12} representing their joint behavior.

First, the set of local, global, and history variables are obtained by taking the unions of those of the two modules: $V_{12}^{all} = V_1^{all} \cup V_2^{all}$, $V_{12}^{L} = V_1^{L} \cup V_2^{L}$, and $V_{12}^{H} = V_1^{H} \cup V_2^{H}$. The same stands for the set of actions: $Act_{12}^{G} = Act_1^{G} \cup Act_2^{G}$ and $Act_{12}^{L} = Act_1^{L} \cup Act_2^{L}$. The input and output invariants of M_{12} are obtained by conjoining those of M_1 and M_2, and so for the initial condition.

The most crucial part in the definition of the product concerns the transitions associated to the actions of M_{12}. Those transitions are a suitable combination of the transitions of M_1 and M_2.

Similarly to other interface models, for each shared action, the output transition of M_1 synchronizes with the input transition of M_2, and symmetrically, the output transition of M_2 is synchronized with the input transition 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 the same shared action are also synchronized, and lead to an input transition in the product. Output transitions, on the other hand, are not synchronized between them: 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 modules interleave asynchronously on transitions labeled by non-shared actions. We now describe in more details the interleaving on shared actions.

If M_1 has an input transition $\rho_1^I(a)$, and M_2 has an input transition $\rho_2^I(a)$, then M_{12} has an input transition $\rho_{12}^I(a)$. The local and global part of $\rho_{12}^I(a)$ are obtained by conjoining those of ρ_1^I and ρ_2^I, i.e., $\rho_{12}^{IL}(a) = \rho_1^{IL}(a) \wedge \rho_2^{IL}(a)$ and $\rho_{12}^{IG}(a) = \rho_1^{IG}(a) \wedge \rho_2^{IG}(a)$. This models the fact that M_1 and M_2 can react jointly to inputs from the environment.

The situation is more complicated for output transitions. Suppose that M_1 has an output transition $\rho_1^O(a)$, and M_2 has an input transition $\rho_2^I(a)$. The result of the two transitions is an output transition $\rho_{12}^O(a)$ in M_{12}, obtained by conjoining $\rho_2^{IL}(a)$ with $\rho_1^O(a)$.

The reader could wonder why the new output transition is not obtained by conjoining also $\rho_2^{IG}(a)$ with $\rho_1^O(a)$. The reason is the definition of input and output transitions: output transitions can modify global variables, while input transitions can only make assumptions on them. The assumptions expressed by the global section of input rules will be taken into account in the next phase of composition.

5.3 Locally Incompatible States

The product defined in the previous section can contain *locally incompatible states*. In a locally incompatible state, one of the modules being composed wants to issue

an output transition labeled by a shared action, while the other module does not have a corresponding global input transition from that state which agrees with the output transition on the updates of global variables. In practice, TICC computes the set of good states *Good*, which is simply the complement of the set of locally incompatible states.

Example 4. Consider the fire detector example of Section 2, illustrated in Figure 2(a). In the composition of `ControlUnit` and `Faulty_FireDetector2`, the state where `ControlUnit.s` $= 3$ and `Faulty_FireDetector2.s` $= 1$ is locally incompatible: module `Faulty_FireDetector2` can issue the output action `fire`, which module `ControlUnit`, being disabled, cannot accept.

5.4 Synthesizing a Strategy

After computing the product of the two modules and the set of good states, the next operations is to compute the set of states *Win* from which the Input player of M_{12} has a strategy to always stay in *Good*. This is done by playing a safety game whose objective is *Good*. The result of the game is used to restrict the input invariant of the product (use the command `print_input_restriction` to see how the new invariant restrict the Input transitions of the composition). Hence the composition of the two modules can only works in environments that satisfy the restricted input invariant. This can be considered an optimistic approach, since two modules are not considered to be incompatible if they cannot work in one particular environment.

The set *Win* is also conjoined with the initial condition of the product, giving rise to the initial condition of the composition. If the resulting initial condition is empty, the two modules are definitely incompatible.

Example 5. Consider again the fire detector example of Section 2, illustrated in Figure 2(a). The modules `ControlUnit` and `Faulty_FireDetector2` are compatible: in fact, there is an environment that avoids all locally incompatible states. For instance, to avoid the state where `ControlUnit.s` $= 3$ and `Faulty_FireDetector2.s` $= 1$, the environment can simply avoid issuing the action `smoke2` if `disable` has already been issued, or can avoid to issue action `disable` if `smoke2` has already been issued.

Of course, such a compatibility masks the fact that it does not make sense to restrict the environment's ability to issue actions `smoke2` — a fire can start at any time! The user can discover the problem by asking TICC to print the *restriction* of action `smoke2`, via the command

```
print_input_restriction d "smoke2";;
```

which generates the following output:

```
Restriction of input action smoke2:
(
 (Faulty_FireDetector2.s = 0)(
   (ControlUnit.s = 3)) )
```

This indicates that, after the composition, action `smoke2` can no longer be accepted if no smoke has been detected yet (`Faulty_FireDetector2.s = 0`) and the controller has been disabled (`ControlUnit.s = 3`).

Similarly, the user can print the restriction of action `disable` in the composition of `ControlUnit` and `Faulty_FireDetector2` to discover how the ability of accepting `disable` has been restricted by the composition.

6 Composition: A Concrete Example

In this section we present a concrete example of the use of TICC on a large program. We consider a model of the interaction among contractors fixing a house. The example illustrates how TICC can verify the compatibility of the interaction protocol among communicating entities.

The example models a house with four rooms: a K(itchen), a L(iving), a B(athroom), and a (Bed) R(oom). Each room can suffer from electrical and plumbing problems that can be fixed by a plumb(er) and an electr(ician). Depending of the problem that occurred, contractors are also needed to repair the damages caused on the wall and on the floor. After the repairs, the room has to be cleaned. As rooms are small, only one contractor at a time can work in a room.

We wish to know if the contractors can work together and fix the problems. This question can be answered in TICC by modeling each contractor as a module, and by considering additional modules that simulate faults, and that call the contractors to fix things. The contractors can work together if the composition of all the modules is compatible.

The TICC program corresponding to the example is as follows. Each room may have ongoing repair work; this is tracked by the following global variables:

```
var K_busy, L_busy, B_busy, R_busy: bool
```

In each room, four items might need repair: plumb(ing), electr(ical), floor, and wall. Moreover, the room may need to be clean(ed). For the kitchen, the need for repair and the need to clean are tracked by the following global variables (where a `true`variable means that the corresponding item is broken):

```
var K_plumb, K_electr, K_floor, K_wall, K_clean: bool
```

Similar variables track the state of L(iving room), B(athroom), and (bed)R(oom). The activity state of the five contractors is tracked by the following global variables:

```
var plumb_active, electr_active, floor_active,
    wall_active, clean_active: bool
```

At the start, one supposes that there is no ongoing work in the room, meaning that the contractors are not working.

```
stateset initcond: ~K_busy & ~L_busy & ~B_busy & ~R_busy & ~plumb_active
         & ~electr_active & ~floor_active & ~wall_active & ~clean_active
```

After these declarations, we declare the modules. The module `Breaks` models plumbing and electrical failures. The code for this module is given in Figure 7. The body of the module contains a series of declarations of output transitions. As an example, the

```
1  module Breaks:
2    stateless
3          K_plumb, K_electr, K_floor, K_wall, K_clean,
4          L_plumb, L_electr, L_floor, L_wall, L_clean,
5          B_plumb, B_electr, B_floor, B_wall, B_clean,
6          R_plumb, R_electr, R_floor, R_wall, R_clean
7
8    output break_K_plumb : { ~K_plumb ==> K_plumb' & K_floor' & K_wall'
         & K_clean' }
9    output break_L_plumb : { ~L_plumb ==> L_plumb' & L_floor' & L_wall'
         & L_clean' }
10   output break_B_plumb : { ~B_plumb ==> B_plumb' & B_floor' & B_wall'
         & B_clean' }
11   output break_R_plumb : { ~R_plumb ==> R_plumb' & R_floor' & R_wall'
         & R_clean' }
12
13   output break_K_electr : { ~K_electr ==> K_electr' & K_wall' &
         K_clean' }
14   output break_L_electr : { ~L_electr ==> L_electr' & L_wall' &
         L_clean' }
15   output break_B_electr : { ~B_electr ==> B_electr' & B_wall' &
         B_clean' }
16   output break_R_electr : { ~R_electr ==> R_electr' & R_wall' &
         R_clean' }
17 endmodule
```

Fig. 7. Module `Breaks` for the house example.

following transition models the fact that, when the plumbing in the kitchen is not broken (~ means "not", and `K_plumb` tracks whether the kitchen plumbing works), then it can break, generating the output transition `break_K_plumb`, and signaling that the kitchen plumbing, floor, and walls need repair. Moreover, the room needs to be cleaned.

```
output break_K_plumb : { ~K_plumb ==> K_plumb' & K_floor' &
                         K_wall' & K_clean'}
```

All global variables are history free for this module.

The module `Calls` calls the repairmen and the cleaner when needed (the code of this module is given in Figure 8); as an example, the plumber is called using the following statement:

```
output call_plumb : { ~plumb_active &
       (K_plumb | L_plumb | B_plumb | R_plumb) ==> plumb_active' }
```

Note that all the variables are history free for this module. This choice is quite obvious since, as an example, there is no reason for `Calls` to track the value of `plumb_active` after it has called the plumber. If global variables where not history free, then one would be forced to add many new input rules to the module.

```
1  module Calls:
2    stateless
3          K_plumb, K_electr, K_floor, K_wall, K_clean,
4          L_plumb, L_electr, L_floor, L_wall, L_clean,
5          B_plumb, B_electr, B_floor, B_wall, B_clean,
6          R_plumb, R_electr, R_floor, R_wall, R_clean,
7          plumb_active, electr_active, floor_active, wall_active,
               clean_active
8
9    output call_plumb : { ~plumb_active & (K_plumb | L_plumb | B_plumb |
         R_plumb ) ==> plumb_active' }
10   output call_electr : { ~electr_active & (K_electr | L_electr |
         B_electr | R_electr) ==> electr_active' }
11   output call_floor : { ~floor_active & (K_floor | L_floor | B_floor |
         R_floor ) ==> floor_active' }
12   output call_wall : { ~wall_active & (K_wall  | L_wall  | B_wall  |
         R_wall ) ==> wall_active' }
13   output call_clean : { ~clean_active & (K_clean | L_clean | B_clean |
         R_clean ) ==> clean_active' }
14
15 endmodule
```

Fig. 8. Module `Calls` for the house example. The module calls the repairmen and the cleaner.

After the declaration of the modules `Breaks` and `Calls`, come the declarations of the modules for the five contractors.

The plumber, whose part of the code is given in Figure 9, and the other contractors keep track of whether they are working via a Boolean variable `working`. Also, they keep track of the room on which they are working via the local Boolean variables `Kw`, `Lw`, `Bw`, `Rw`. W hen called, the plumber is initially not working on any room.

```
input call_plumb : { local: ~plumb_active ==> working' := false }
```

When an active plumber, not working on any room, sees that the K(itchen) is unoccupied (`~K_busy`) and needs repair (`K_plumb`), the plumber starts to work in the K(itchen):

```
output K_start_plumb :
 { plumb_active & ~working & K_plumb & ~K_busy
   ==>
   working' & Kw' & K_busy' }
```

and similarly for the other rooms.

While working in the kitchen, the plumber does not expect anybody else to work in it. Thus, we have to define input transitions corresponding to the actions of the other contractors. As an example, the following rule forbids the electrician to start working in the kitchen if the plumber is still working there.

```
input  K_start_electr : { local: ~Kw ==>  }
```

```
1  module Plumber:
2    var working: bool
3    var Kw, Lw, Bw, Rw: bool
4    initial: ~working & ~Kw & ~Lw & ~Bw & ~Rw
5    stateless
6          K_plumb, K_electr, K_floor, K_wall, K_clean,
7          L_plumb, L_electr, L_floor, L_wall, L_clean,
8          B_plumb, B_electr, B_floor, B_wall, B_clean,
9          R_plumb, R_electr, R_floor, R_wall, R_clean
10
11   input call_plumb : {local: ~plumb_active ==> working' := false }
12   output done_plumb : { plumb_active & ~working & ~K_plumb & ~L_plumb
         & ~B_plumb & ~R_plumb ==> ~plumb_active' }
13
14
15   output K_start_plumb : { plumb_active & ~working & K_plumb & ~K_busy
          ==> working' & Kw' & K_busy' }
16   output L_start_plumb : { plumb_active & ~working & L_plumb & ~L_busy
          ==> working' & Lw' & L_busy' }
17   output B_start_plumb : { plumb_active & ~working & B_plumb & ~B_busy
          ==> working' & Bw' & B_busy' }
18   output R_start_plumb : { plumb_active & ~working & R_plumb & ~R_busy
          ==> working' & Rw' & R_busy' }
19   output K_done_plumb : { plumb_active & Kw ==> ~K_plumb' & ~Kw' & ~
         K_busy' & ~working' }
20   output L_done_plumb : { plumb_active & Lw ==> ~L_plumb' & ~Lw' & ~
         L_busy' & ~working' }
21   output B_done_plumb : { plumb_active & Bw ==> ~B_plumb' & ~Bw' & ~
         B_busy' & ~working' }
22   output R_done_plumb : { plumb_active & Rw ==> ~R_plumb' & ~Rw' & ~
         R_busy' & ~working' }
23
24   input K_* : { local: ~Kw ==> }
25   input L_* : { local: ~Lw ==> }
26   input B_* : { local: ~Bw ==> }
27   input R_* : { local: ~Rw ==> }
28 endmodule
```

Fig. 9. Module describing the plumber.

One of the main drawbacks of this formalization is that we have to define many input transitions that differ only by their name but not by their contents. To simplify the declaration of such inputs, TICC allows the use of wildcard action names. Figure 9 shows how wildcard inputs can simplify the description of the module `Plumber`. Using the special character "`*`", input transition rules can be defined to match a set of actions instead of one action only. For instance, the pattern `K_*` on line 24 of Figure 9 matches any action whose name starts with `K_`.

In module `Plumber`, variable `plumb_active` is a history variable, as the module plans to control its value. Variables `K_busy`, `L_busy`, `B_busy`, and `R_busy` are also history variables. This choice, combined with the declaration of the input transitions, ensures that the value of those variables can be changed by other modules only if the plumber is not working in the corresponding room.

We considered two different electrician modules. A "correct" implementation, `Electrician`, checks that the kitchen is free before starting to work in it:

```
output K_start_electr :
 { electr_active & ~working & K_electr & ~K_busy
   ==>
   working' & Kw' & K_busy' }
```

Note that above, the variable `Kw` is local to the electrician, and indicates whether the electrician is working on the kitchen; the equally-named variable `Kw` in (*) is instead local to the plumber. An "incorrect" implementation of the electrician, `WElectrician`, in the rush of getting things done, forgets to check whether somebody else is already at work in the kitchen:

```
output K_start_electr :
 { electr_active & ~working & K_electr ==> working' & Kw' & K_busy' }
```

TICC is able to detect that the composition of `Breaks`, `Calls`, `Plumber`, and `Electrician` is compatible (see lines from 14 to 16 of Figure 10), whereas it detects that the composition of `Breaks`, `Calls`, `Plumber`, and `WElectrician` is not. Thus, the protocol violation can be discovered before the complete system, consisting also of modules to repair floors and walls, is constructed. In fact, a simple check would have revealed the problem already in the composition of `Plumber` and `WElectrician` (as computed in line 18 of Figure 10). When composing `Plumber` and `WElectrician`, TICC automatically synthesizes the assumption that (i) they are not both called to work, or (ii) no room needs to be repaired by both of them.

We also note that the protocol violation is revealed *thanks to the input assumption of the correct module* `Plumber`. In the game-based approach that underlies TICC, the input assumptions of correct modules constrain the protocol of modules that will be later composed into the system, preventing the composition of "rogue" modules. The verification of the correctness of interaction is simply a by-product of composition. This situation should be contrasted to the usual, non-game-based approach to modeling and verification. In the usual approach, detecting incompatibilities requires writing separate specifications of correctness, and can usually be performed only once all components are composed.

```
 1  open Ticc;;
 2
 3  parse "house.si";;
 4
 5  let breaks     = mk_sym "Breaks";;
 6  let calls      = mk_sym "Calls";;
 7  let plumber    = mk_sym "Plumber";;
 8  let electrician = mk_sym "Electrician";;
 9  let rudelectr  = mk_sym "RudeElectrician";;
10  let floors     = mk_sym "Floors";;
11  let walls      = mk_sym "Walls";;
12  let clean      = mk_sym "Clean";;
13
14  let c0 = compose breaks calls;;
15  let c1 = compose c0 plumber;;
16  let c2 = compose c1 electrician;;
17
18  let d2 = compose c1 rudelectr;;
```

Fig. 10. TICC script for the house example: `house.in`.

7 Additional Tool Features

While composition is certainly the most important operation that TICC can perform on modules, it is not the only one. This section is a brief introduction to the other features of the tool.

7.1 Symbolic Operations, Model Checking, and Simulation

A set of states, in TICC, can be defined via a formula specifying constraints on the values of the variables. TICC can parse such formulas, and construct a symbolic representation (an MDD) that enables it to manipulate the set. TICC can combine such sets with the usual Boolean operators, via the functions `set_or`, `set_and`, `set_implies`, and `set_not`; sets can also be compared using `set_is_subset` and `set_equal`. A set of states can be printed using the command `print_stateset` (printing is not optimized, and can lead to exponentially large printouts). TICC also contains an implementation of the classical CTL operators [9], allowing the user to verify properties of models via model checking. As usual, the CTL operators are documented in `doc/api/Ticc.html`.

Example 6. Consider the fire detection system given in Figure 2(a), and the script file in Figure 11. Line 11 builds the symbolic representation of a set ϕ consisting of the states where `ControlUnit.s = 2`, i.e., the firemen have been called. Line 13 prints the set of states that satisfy the CTL formula $\exists\Diamond\phi$, and line 15 prints the set of states that satisfy the CTL formula $\forall\Diamond\phi$.

TICC can also perform random simulation on symbolic modules, generating an HTML file with the result of the simulation. This is particularly useful in the early stages of model construction, to confirm that the model behaves as intended.

```
1  open Ticc;;
2  parse "fire-detector-disable.si";;
3
4  let fire1 = mk_sym "FireDetector1";;
5  let controlunit = mk_sym "ControlUnit";;
6  let comp = compose fire1 controlunit;;
7
8  let clone_fire1 = sym_clone fire1;;
9  simulate comp "Fire1.s = 0 & ControlUnit.s = 0", 5, "detector.html";;
10
11 let called_firemen = parse_stateset ("ControlUnit.s = 2");;
12 print_string "Can call the firemen:";;
13 print_stateset (ctl_e_f comp called_firemen);;
14 print_string "Always calls the firemen:";;
15 print_stateset (ctl_a_f comp called_firemen);;
```

Fig. 11. A script file illustrating individual operations.

7.2 Closure

TICC allows the user to close a module with respect to the occurrence of input transitions. After several modules have been composed, the closure operation can be used to say that the environment is no longer able to provide a certain input. The following example illustrates the use of the closure operation in the context of CTL model checking.

Example 7. We consider a simple dining philosophers model, where n philosophers are sitting at a round table. Set between each pair of neighboring philosophers are n forks, so that all philosophers have a fork on their left, and one on their right. Each philosopher can either think or try to eat. To be able to eat, philosophers, being rather clumsy, have to use both forks on their sides.

Each philosopher `Phil` can be in one of 7 internal states that are enumerated with a local variable `s`. In `s=0`, `Phil` is thinking; a transition to `s=1` indicates the philosopher's desire for food. In state `s=4` the philosopher eats. To go from `s=1` to `s=4`, `Phil` has to grab the two forks. This can be done in any order (requiring the addition of two intermediate states `s=2` and `s=3`, depending on which fork has been chosen first). After having eaten, `Phil` releases the forks in nondeterministic order, and starts thinking again.

The TICC program of Figure 12 and its corresponding script file given in Figure 13 show an example of dining philosophers with $n = 2$ philosophers and thus 2 forks. The program can easily be extended to a greater number of philosophers. In the program, the philosophers are represented by modules `Phil1` and `Phil2`, while the forks with Boolean global variables `F1`, `F2`, and `F3`, whose value is `true` if the fork is available, and `false` otherwise. The actions of grabbing and releasing forks are modeled by the actions `GrabF`x and `givebackF`x, where $x \in \{1,2\}$ identifies the fork. Since a fork is shared between two philosophers, each philosopher must both output these actions, and

```
1  var F1, F2: bool
2  stateset initcond: F1 & F2
3
4  module Phil1:
5        var s: [0..6]
6        initial: s = 0
7
8        input *: {}
9        local no_moves: { true ==> }
10       local wants_to_eat: { s = 0 ==> s' = 1 }
11       output grabF1: { s = 1 & F1 ==> s' = 2 & ~F1';
12             s = 3 & F1 ==> s' = 4 & ~F1' }
13       output grabF2: { s = 1 & F2 ==> s' = 3 & ~F2';
14             s = 2 & F2 ==> s' = 4 & ~F2' }
15       output givebackF1: { s = 4 ==> s' = 5 & F1';
16             s = 6 ==> s' = 0 & F1' }
17       output givebackF2: { s = 4 ==> s' = 6 & F2';
18             s = 5 ==> s' = 0 & F2' }
19 endmodule
20
21 module Phil2:
22       var s: [0..6]
23       initial: s = 0
24
25       input *: {}
26       local no_moves: { true ==> }
27       local wants_to_eat: { s = 0 ==> s' = 1 }
28       output grabF2: { s = 1 & F2 ==> s' = 2 & ~F2';
29             s = 3 & F2 ==> s' = 4 & ~F2' }
30       output grabF1: { s = 1 & F1 ==> s' = 3 & ~F1';
31             s = 2 & F1 ==> s' = 4 & ~F1' }
32       output givebackF2: { s = 4 ==> s' = 5 & F2';
33             s = 6 ==> s' = 0 & F2' }
34       output givebackF1: { s = 4 ==> s' = 6 & F1';
35             s = 5 ==> s' = 0 & F1' }
36 endmodule
```

Fig. 12. A TICC dining philosophers model: `dining.si`.

```
 1  open Ticc;;
 2
 3  parse "phil.si";;
 4
 5  let phil1 = mk_sym "Phil1";;
 6  let phil2 = mk_sym "Phil2";;
 7  let comp_phils = compose phil1 phil2;;
 8
 9  let initial = parse_stateset "Phil1.s = 0 & Phil2.s = 0 & F1 & F2 ";;
10  let bad_fork = parse_stateset "Phil1.s = 0 & Phil2.s = 0 & ~F2";;
11
12  let can_reach_bad_fork_exists = ctl_e_f comp_phils bad_fork;;
13  let result = set_and can_reach_bad_fork_exists initial;;
14  print_stateset result;;
15
16  let comp_phils_close = close comp_phils "*";;
17
18  let can_reach_bad_fork_exists = ctl_e_f comp_phils_close bad_fork;;
19  let result = set_and can_reach_bad_fork_exists initial;;
20  print_stateset result;;
```

Fig. 13. A TICC script for the dining philosophers.

be able to accept them as input from other philosophers. This is the purpose of the wildcard input `input *`.

The problem is that, once `Phil1` and `Phil2` are composed, their composition can *still* accept the actions `GrabF`*x* and `givebackF`*x* from the environment. It is as if passers-by were allowed to pick up and put down forks! Indeed, in the composition of `Phil1` and `Phil2`, we can start from the state where `Phil1` and `Phil2` are both thinking and `F2` is available and reach a state where the philosophers are still thinking but `F2` is not available, as it has been "picked up" by the environment. This is shown by the fact that the stateset printed at line 14 is not empty.

This clearly does not make sense: once `Phil1` and `Phil2` are composed, we should be able to say that the forks are no longer in the environment's reach. To this end, we *close* the composition of `Phil1` and `Phil2` with respect to all input actions.[8] Once this is done, the state where both philosophers are thinking but `F2` is not available is no longer reachable, and indeed the printout from line 20 is the empty set (represented as `(0)`).

8 Conclusions

Interface theories are the subject of many recent works. The sociable interface model presented in this paper is only one of them. Interface models that appeared before socia-

[8] In general, we can close a module with respect to any set of actions.

ble interfaces include interface automata [13, 15] and interface modules [14, 8]. Those models were based on a communication with 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 in the communication process, they take advantage of the existing models and provide rich communication primitives.

The tool TICC is certainly not the first tool that implements an interface model, and even not the most complete. As an example, the tool CHIC that implements a synchronous, variable-based interface theory is able to handle pushdown games while TICC cannot.

However, one major difference between TICC and its predecessors is its ability to use rich communication primitives to model components in a very compact and natural way. Another strong point of the tool is its symbolic implementation which makes it very efficient and easily extensible.

TICC is a tool in constant evolution, and so is the sociable interface model. As an example, we are currently developing a real-time extension of the tool, based on the *Timed Interfaces* of [16]. 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. Another direction we are considering is the implementation of the alternating-time temporal logic of [2]. This logic is more suitable to model check open systems than CTL.

References

1. 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 *Lecture Notes in Computer Science*. Springer, 2006. to appear.
2. 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.
3. R. Alur, T.A. Henzinger, F.Y.C. Mang, S. Qadeer, S.K. Rajamani, and S. Tasiran. Mocha: modularity in model checking. In *CAV 98: Proc. of 10th Conf. on Computer Aided Verification*, volume 1427 of *Lect. Notes in Comp. Sci.*, pages 521–525. Springer-Verlag, 1998.
4. R.E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, 1986.
5. A. Chackrabarti, L. de Alfaro, T.A. Henzinger, and M. Stoelinga. Resource interfaces. In *EMSOFT 03: 3rd Intl. Workshop on Embedded Software*, volume 2855 of *Lect. Notes in Comp. Sci.*, pages 117–133. Springer-Verlag, 2003.
6. A. Chackrabarti, L. de Alfaro, M. Jurdziński, K. Chatterjee, T.A. Henzinger, and F.Y.C. Mang. CHIC: Checker for interface compatibility, 2003. www-cad.eecs.berkeley.edu/ tah/chic/.
7. A. Chakrabarti, L. de Alfaro, T.A. Henzinger, Marcin Jurdziński, and F.Y.C. Mang. Interface compatibility checking for software modules. In *CAV 02: Proc. of 14th Conf. on Computer Aided Verification*, volume 2404 of *Lect. Notes in Comp. Sci.*, pages 428–441. Springer-Verlag, 2002.
8. 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.
9. E.M. Clarke, O. Grumberg, and D.A. Peled. *Model Checking*. MIT Press, 1999.

10. L. de Alfaro. Game models for open systems. In *Proceedings of the International Symposium on Verification (Theory in Practice)*, volume 2772 of *Lect. Notes in Comp. Sci*. Springer-Verlag, 2003.
11. L. de Alfaro, R. Alur, R. Grosu, T. Henzinger, M. Kang, R. Majumdar, F. Mang, C. Meyer-Kirsch, and B.Y. Wang. Mocha: A model checking tool that exploits design structure. In *ICSE 01: Proceedings of the 23rd International Conference on Software Engineering*, 2001.
12. 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 *Lecture Notes in Computer Science*, pages 81–105. Springer, 2005.
13. 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.
14. 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.
15. 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.
16. 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.
17. 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.
18. M. Faella and A. Legay. Some models and tools for open systems. Technical report, University of Santa Cruz, 2005. Proceedings of FIT05.
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. N.A. Lynch. *Distributed Algorithms*. Morgan-Kaufmann, 1996.
22. Fabio Somenzi. Cudd: Cu decision diagram package. http://vlsi.colorado.edu/ fabio/CUDD/cuddIntro.html.
23. A. Srinivasan, T. Kam, S. Malik, and R. Brayton. Algorithms for discrete function manipulation. In *Proceedings International Conference CAD (ICCAD-91)*, 1990.